

Федоренко Ю.

Алгоритмы и программы на *C++Builder*



Москва, 2010

УДК 004.4
ББК 32.973.26-018.2
Ф33

Ф33 Федоренко Ю. П.

Алгоритмы и программы на *C++Builder*. – М.: ДМК Пресс, 2010. – 544с.: ил.

ISBN 978-5-94074-607-2

В книге подробно рассмотрены синтаксис, семантика, техника процедурного и объектно-ориентированного программирования на *C++Builder*. Язык *C++* и базовые алгоритмы обработки данных всех типов изучаются параллельно с визуальным программированием. Книга будет также полезна тем, кто имеет определённый опыт в программировании, однако желает расширить и углубить свои знания. Она может служить надёжной платформой для изучения и других языков программирования, поскольку она основывается на парадигме разработки современных приложений, к которой специалисты продвигались более пяти десятилетий.

Издание предназначено для студентов, преподавателей, инженеров, научных сотрудников, лицеистов и старшеклассников, желающих самостоятельно изучить *C++Builder* «с нуля».

К книге прилагается компакт-диск, на котором записаны примеры программ для среды *C++ Builder 5* и *C++ Builder 6*.

УДК 004.4
ББК 32.973.26-018.2

Федоренко Юрий Петрович
Алгоритмы и программы на *C++Builder*

Главный редактор Мовчан Д. А.
dm@dmk-press.ru
Корректор Синяева Г. И.
Верстка Паранская Н. В.
Дизайн обложки Мовчан А. Г.

Подписано в печать 16.12.2009. Формат 70x100 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 36. Тираж 1000 экз.

Web-сайт издательства: www.dmk-press.ru
Internet-магазин: www.aliants-kniga.ru

ISBN 978-5-94074-607-2

© Федоренко Ю. П., 2010
© Оформление, ДМК Пресс, 2010



Содержание

Благодарности	11
Предисловие	12
Урок 1. Первая программа	20
1.1. Внешний вид и назначение приложения <i>Умножитель</i>	20
1.2. Знакомство с визуальным программированием	21
1.2.1. Форма	22
1.2.2. Тип объекта	23
1.2.3. Правильный идентификатор	24
1.2.4. Поле ввода	25
1.2.5. Поле вывода	27
1.2.6. Командная кнопка	28
1.3. Разработка кода программы	28
1.3.1. Что такое алгоритм	28
1.3.2. Алгоритм программы	29
1.3.3. Способы порождение заготовки функции	30
1.3.4. Заголовок функции	30
1.3.5. Задание переменных	32
1.3.6. Однострочный комментарий	33
1.3.7. Инициализация переменных	33
1.3.8. Конфигурация компьютера	34
1.3.9. Оператор присваивания	34
1.3.10. Текстовая константа	35
1.3.11. Многострочный комментарий и преобразование текстовых переменных в переменные вещественного типа	35
1.3.12. Преобразование переменных вещественного типа в переменные текстового типа	36
1.3.13. Форматирование текста программы	37
1.4. Сохранение и отладка	38
1.4.1. Сохранение проекта	38
1.4.2. Компиляция и сборка проекта	39
1.4.3. Предупреждения и подсказки	43
1.4.4. Запуск проекта	43
1.5. Задача для программирования	44
1.6. Вариант программного решения	45

Урок 2. Модернизация программы	47
2.1. Открытие проекта	47
2.2. Округление результата	47
Приложение 2.1. Возможности функции <i>FloatToStrF()</i>	48
2.3. Борьба с ошибками пользователя	49
2.3.1. Исключительные ситуации	49
2.3.2. Молчаливые исключения	54
2.4. Улучшение интерфейса	55
2.4.1. Выбор пиктограммы для приложения	55
2.4.2. Местоположение интерфейса	55
2.4.3. Плавное перетаскивание и протягивание	55
2.4.4. Горизонтальное и вертикальное выравнивание	56
2.4.5. Выравнивание по сетке	57
2.4.6. Уравнивание габаритов	58
2.4.7. Одинаковые интервалы	59
2.4.8. Симметрирование объектов относительно центра	59
2.4.9. Порядок перехода между объектами управления	60
2.5. Подробнее об отладке программы	60
2.5.1. Предупреждения и подсказки	60
2.5.2. Точки остановки и всплывающая подсказка	62
2.5.3. Пошаговое выполнение программы	64
2.6. Полноценный исполняемый файл	65
2.7. Задача для программирования	68
2.8. Вариант программного решения	68
 Урок 3. Операции, математические функции и операторы выбора	 70
3.1. Использование математических функций и констант	70
3.1.1. Условие задачи и интерфейс программы <i>Дальность</i>	70
3.1.2. Препроцессорная обработка	71
3.1.3. Файл реализации	72
3.1.4. Описание программы	72
3.2. Основные стандартные математические функции	74
3.3. Операции и порядок их выполнения	76
3.3.1. Операции отношения	76
3.3.2. Круглые скобки	76
3.3.3. Логические операции	77
3.3.4. Арифметические операции	78
3.3.5. Операции присваивания	79
3.3.6. Порядок вычислений	80
3.4. Условный оператор <i>if</i>	80
3.5. Оператор множественного выбора <i>switch</i>	85
3.6. Дополнительные возможности командной кнопки	88
3.7. Задача для программирования	89
3.8. Вариант решения задачи	89

Урок 4. Все о циклах	90
4.1. Зачем нужны циклы?	90
4.2. Оператор цикла <i>for</i>	96
4.3. Оператор цикла <i>while</i>	103
4.4. Оператор цикла <i>do_while</i>	105
4.5. Вложенные циклы	106
4.6. Задачи для программирования	112
4.7. Варианты решений задач	113
 Урок 5. Графики зависимостей	 117
5.1. Построение одномерных зависимостей	117
5.2. Построение серии одномерных графиков	127
5.3. Модернизируем интерфейс	130
5.3.1. Кнопка для выхода из приложения	130
5.3.2. Показ стандартных мультфильмов	132
5.3.3. Стандартные кнопки <i>Windows</i>	134
5.3.4. Размещение картинок на кнопках	135
5.3.5. Размер формы и граничные пиктограммы	135
5.4. Задачи для программирования	136
5.5. Варианты решений задач	137
 Урок 6. Одномерные массивы	 139
6.1. Зачем нужны массивы?	139
6.1.1. Интерфейс приложения	140
6.1.2. Задание массива	141
6.1.3. Суммирование элементов массива	143
6.1.4. Определение экстремума	144
6.1.5. Счетчик	145
6.1.6. Функция <i>PyskClick</i> в законченном виде	146
6.1.7. Компактный вариант функции <i>PyskClick</i>	147
6.1.8. Общие определения	148
6.1.9. Глобальные переменные и константы	149
6.1.10. Страж кода	150
6.2. Метод линейной сортировки	152
6.2.1. Сущность метода	152
6.2.2. Алгоритм метода	153
6.2.3. Ручная трассировка алгоритма	154
6.2.4. Иллюстрация метода	156
6.3. Сортировка методом пузырька	159
6.4. Другие методы упорядочивания элементов массива	162
6.4.1. Применение трёх массивов	162
6.4.2. Применение одного массива	163
6.4.3. Упорядочивание массива методом <i>Ларионова</i>	166
6.5. Схемы алгоритмов	167
6.6. Задачи для программирования	172
6.7. Варианты решений задач	172

Урок 7. Многомерные массивы	178
7.1. Примеры многомерных массивов	178
7.2. Описание многомерных массивов	179
7.3. Доступ к элементам массива	181
7.4. Главная и побочная диагонали	182
7.5. Примеры обработки матриц	183
7.6. Задачи для программирования	189
7.7. Варианты решений задач	189
 Урок 8. Функции	 194
8.1. Важный инструмент структурирования программ	194
8.2. Что уже известно о функциях	195
8.3. Описание и объявление функций	196
8.3.1. Тип возвращаемых значений	196
8.3.2. Если функция не имеет параметров	198
8.3.3. Прототип функции	198
8.3.4. Переменное число параметров функции	199
8.3.5. Параметры со значениями по умолчанию	200
8.3.6. Чем функции отличаются от программ	200
8.3.7. Операция расширения области видимости	200
8.3.8. Выход из функции	201
8.3.9. Запрет и разрешение доступа к функции	201
8.4. Различные способы передачи параметров в функции	202
8.4.1. Передача параметра по значению	202
8.4.2. Передача параметра по ссылке	203
8.4.3. Передача параметра по адресу	204
8.4.4. Применение спецификатора <i>const</i> в ссылочных параметрах	205
8.4.5. Применение спецификатора <i>const</i> в параметрах-указателях	205
8.5. Перегрузка функций	206
8.6. Шаблоны функций	207
8.6.1. Параметры одинакового типа	208
8.6.2. Параметры разного типа	208
8.6.3. Параметры разного формального типа и конкретного типа	208
8.6.4. Необходимость применения всех типов объявленных параметров	209
8.6.5. Другие возможности и ограничения шаблонов	209
8.6.6. Пример практического применения шаблона	210
8.7. Иллюстрация применения пользовательских функций	212
8.8. Задача для программирования	219
8.9. Вариант решения задачи	219
 Урок 9. Файлы	 224
9.1. Назначения файлов	224
9.1.1. Устройства для долговременного хранения информации. История вопроса	224
9.1.2. Что называется файлом	225

9.1.3. Что содержится в файлах	226
9.1.4. Типы файлов	227
9.1.5. Работа с файлами на физическом уровне	227
9.2. Простые приложения с текстовыми файлами	229
9.2.1. Многострочные окна редактирования <i>Memo</i> и <i>RichEdit</i>	229
9.2.2. Невизуальный объект типа <i>TStringList</i>	231
9.3. Обработка массивов в текстовых файлах	234
9.3.1. Чтение из файла и запись в файл одномерного массива	234
9.3.2. Чтение из файла и запись в файл двумерных массивов	236
9.4. Двоичные файлы	240
9.4.1. Простое приложение с двоичным файлом	243
9.4.2. Одномерный массив в двоичном файле	243
9.4.3. Матрица в двоичном файле	244
9.4.4. Приложение <i>Квадрат числа</i>	245
9.5. Улучшаем интерфейс приложения	247
9.5.1. Создаём меню	248
9.5.2. Создаём горячие клавиши	251
9.5.3. Создаём кнопки быстрого доступа	251
9.6. Задачи для программирования	256
9.7. Варианты решений задач	256

Урок 10. Указатели и динамические переменные 261

10.1. Вводные замечания и основные определения	261
10.2. Иллюстрация применения указателей	265
10.3. Указатели на различные типы данных	266
10.3.1. Виды указателей	266
10.3.2. Операции с однотипными и разнотипными указателями	267
10.3.3. Указатель на указатель	268
10.3.4. Константный указатель на константные данные	269
10.3.5. Неконстантный указатель на константные данные	269
10.3.6. Константный указатель на неконстантные данные	270
10.4. Адресная арифметика с массивами	270
10.5. Адресная арифметика с указателями на массивы	272
10.6. Массивы указателей	274
10.7. Динамические массивы	279
10.7.1. Одномерные динамические массивы	279
10.7.2. Двумерные динамические массивы	282
10.8. Указатели на функции	285
10.9. Функции, возвращающие указатель на массив	289
10.10. Важная рекомендация	290
10.11. Задачи для программирования	291
10.12. Варианты решений задач	291

Урок 11. Ссылки 294

11.1. Основные свойства	294
11.2. Использование модификатора <i>const</i>	297

11.3. Ссылка на функцию	297
11.4. Функции, возвращающие ссылку	299
11.5. Ссылки на объекты динамических переменных	300
11.6. Осваиваем новые объекты интерфейса	301
11.6.1. Переключатели	301
11.6.2. Индикаторы	304
11.6.3. Диалоговое окно	307
11.6.4. Диаграммы длительных процессов	310
11.6.5. Непослушная кнопка	315
11.6.6. Модальные и немодальные формы	316
11.7. Задачи для программирования	326
11.8. Варианты решений задач	326

Урок 12. Символы и строки 329

12.1. Символьные переменные	329
12.1.1. Перевод символов <i>DOS</i> в символы <i>Windows</i>	331
12.1.2. Перевод символов <i>Windows</i> в символы <i>DOS</i>	333
12.2. Массивы символов	335
12.2.1. Задание и инициализация	335
12.2.2. Функции для обработки	336
12.3. Переменные типа <i>String</i>	339
12.3.1. Сравнение строк	342
12.3.2. Стандартные функции для обработки строк	343
12.3.3. Иллюстрация обработки строк	347
12.4. Задачи для программирования	356
12.5. Варианты решений задач	356

Урок 13. Перечисления, множества, объединения и структуры 363

13.1. Перечислимый тип переменных	363
13.2. Множества	365
13.2.1. Операции над однотипными множествами	368
13.2.2. Внутреннее представление множеств	370
13.2.3. Пример применения множеств	372
13.3. Объединения	374
13.4. Структуры	375
13.4.1. Простые структуры	375
13.4.2. Вложенные структуры	378
13.4.3. Массивы структур	379
13.4.4. Структуры с вариантными полями	380
13.4.5. Методы структур	381
13.4.6. Пример обработки базы данных	382
13.4.7. Наследование структур	390
13.4.8. Защита полей и методов структур	392
13.4.9. Двухнаправленный список структур	394

13.5. Задачи для программирования	405
13.6. Варианты решений задач	405

Урок 14. Классы 414

14.1. Вводные замечания	414
14.2. Инкапсуляция и наследование	415
14.3. Новый вариант приложения	422
14.4. Полиморфизм	425
14.4.1 Совместимость объектных типов	425
14.4.2. Полиморфизм, виртуальные методы, раннее и позднее связывание	426
14.4.3. Чисто виртуальные методы, абстрактные классы и полиморфные функции	429
14.4.4. Применение виртуальных и чисто виртуальных методов, правила их описания и использования	429
14.4.5. Преимущества и недостатки виртуальных методов	431
14.5. Клиенты и дружественные функции класса	431
14.6. Статические поля и статические константы класса	432
14.7. Конструкторы класса	435
14.7.1. Конструкторы простых классов	436
14.7.2. Конструкторы производных классов	439
14.7.3. Свойства конструкторов и правила их разработки	443
14.8. Деструктор класса	444
14.8.1. Обычные деструкторы	444
14.8.2. Виртуальные деструкторы	448
14.8.3. Свойства деструкторов и правила их разработки	451
14.9. Создание копий объектов	452
14.9.1. Побитовое копирование	452
14.9.2. Запрет неявного преобразования типов	454
14.9.3. Конструктор копирования	454
14.10. Указатель <i>this</i>	457
14.10.1. Назначения и возможности указателя <i>this</i>	457
14.10.2. Программное порождение визуальных объектов	459
14.11. Локальные классы	463
14.12. Шаблоны классов	463
14.13. Отличие структур и объединений от классов	467
14.14. Задача для программирования	469
14.15. Вариант решения задачи	469

Урок 15. Графика и мультипликация 472

15.1. Основные определения	472
15.2. Логотип приложения	474
15.3. Приложение для просмотра графических файлов	477
15.4. Типы графических файлов	478
15.5. Объекты для хранения изображений, открытие, сохранение и переименование файлов	480

15.6. Графические примитивы и инструменты для их построения	484
15.6.1. Карандаш и кисть	484
15.6.2. Прямоугольник	486
15.6.3. Эллипс	488
15.6.4. Дуга	489
15.6.5. Сектор	489
15.6.6. Линия	490
15.6.7. Ломаная линия	491
15.6.8. Многоугольник	492
15.6.9. Текст	493
15.7. Приложение <i>Пособие</i>	495
15.8. Мультипликация с использованием графических примитивов	500
15.9. Мультипликация с применением битовых образов	504
15.9.1. Использование файлов типа <i>bmp</i>	504
15.9.2. Применение ресурсов программы	508
15.10. Разработка мультфильма <i>Аквариум</i>	513
15.11. Вариант мультфильма <i>Аквариум</i> , разработанный на основе применения пользовательских классов	520
15.12. Задачи для программирования	524
15.13. Варианты решения задач	525

Урок 16. Запуск чужих программ из вашего приложения 527

16.1. Технология <i>OLE</i>	527
16.1.1. Знакомство на примере программы	528
16.1.2. Программное внедрение <i>OLE</i> -объектов	535
16.1.3. Быстрый способ внедрения и связывания объектов на основе существующих документов	536
16.1.4. Сохранение документов в исходных форматах	537
16.1.5. Развитие технологии <i>OLE</i>	538
16.2. Задача для программирования	540
16.3. Вариант решения задачи	540

Список литературы 541

Алфавитный указатель компонентов библиотеки *VCL* 542

Алфавитный указатель функций библиотек *C++Builder*, *API Windows*, директив компилятору, типов данных, операций и других ключевых слов 543



Благодарности

На протяжении шести лет, пока создавалась и шлифовалась эта книга, я получал очень доброжелательную поддержку со стороны моих друзей-коллег, которые читали главы по мере их написания и дали целый ряд важных рекомендаций, касающихся содержания и ясности изложения книги. Гаврику А.П. я благодарен также за то, что он расширил опыт применения пособия в учебном процессе Харьковского национального университета радиоэлектроники. Неоценимые советы и пожелания дал зубр программирования математик Милованов Ю.Б., его замечания, аккуратно записанные на полях, я не только полностью учёл, но отдельные его советы по стилю изложения навсегда взял на вооружение при написании научных работ. Особую признательность и благодарность за ценные предложения и замечания выражаю Дорохову В.Л., добровольно взявшему на себя столь большой труд по рецензированию и редактированию первоначального наброска книги. Все названные мои друзья, как и я, – научные сотрудники, поэтому их мнение весьма существенно, ведь книга в первую очередь направлена на удовлетворение программистских запросов научных сотрудников и инженеров. Испытываю определённую неловкость за возложенную на них нагрузку, несмотря на их заверения о том, что книга доставила им удовольствие. Спасибо вам, друзья.



Предисловие

Все вопросы программирования в этом пособии рассматриваются с *самого начала* и *исчерпывающе полно*. Это делает его полезным даже для тех, кто причисляет себя к программистам с определённым опытом. Однако в первую очередь пособие предназначено для научных сотрудников, инженеров и студентов технических высших и средних учебных заведений, которым в ходе основной деятельности требуется уметь самостоятельно разрабатывать программы.

Место C++ Builder среди других языков и сред программирования

C++ Builder – это одна из самых современных и эффективных сред программирования. Что собой представляет среда программирования и язык программирования? Для ответа на эти и другие близкие вопросы вначале кратко рассмотрим историю развития программирования.

В конце 40-х годов прошлого столетия, когда появились первые ЭВМ (электронно-вычислительные машины), использовалось программирование в адресах: последовательность команд процессора в машинных кодах. Машинный код – это совокупность операций, которые может выполнять ЭВМ. Машинный код иными словами это язык машины. ЭВМ (а позже и персональный компьютер) воспринимает все команды *только* в двоичной системе счисления. При программировании в кодах команды (они значительно отличались на разных ЭВМ) для удобства записывали в восьмеричной системе счисления (получалась более короткая вереница цифр команды). В конечном итоге программа представляла собой набор двоичных цифр. Для выполнения, например, какого-то арифметического действия над двумя числами требовалось обратиться к ячейкам (адресам), где хранятся эти числа и применить к ним код операции (специально выделенный номер), а затем результат операции занести в ячейку по определённому адресу (номеру). В целом этот кошмар, по-другому такой способ программирования называть трудно, продолжался 5 – 10 лет. Хотя и в середине 70-х ещё выпускались учебники по упомянутому языку программирования, однако многие разработчики программ (но далеко не все) уже переходили на языки, более удобные для человека.

В 50-е годы разработан язык *Assembler* (сборщик), в котором номера команд (операций) процессора заменили словами (или частью слов) английского языка. Этот язык программирования является языком *низкого* уровня, его применение существенно упростило разработку программ (написание, проверку и отладку). У каждого процессора имелся *свой* Ассемблер с разным числом команд. *Assembler*

используется и в настоящее время для написания очень быстродействующих и компактных программ. Однако разрабатывать на нём большие программы практически невозможно.

Создателю программы более удобно весь ход решения задачи записать на своём *человеческом* языке, а не в машинных кодах (пусть даже эти коды и заменены мнемоническими, легко запоминающимися обрывками слов). Языки программирования, где используются команды в виде слов, называются языками *высокого* уровня. Поскольку ЭВМ изначально создана для проведения *научных* вычислений, то для удобства «перевода» формул на язык машины первым (1954–57 г.г.) был разработан *Fortran* (*FORmula TRANslator*). Прежде всего, он предназначался научным сотрудникам.

Для профессиональных программистов созданы языки: *Algol* (1958–1960 г.г.), *Cobol* (1959–1960 г.), *C* (1972 г.), *C++* (1983 г.), *Ada* (1983–1995 г.г.), *C++ANSI* (1988 г.), *Object Pascal* (1988 г.) и другие. Если *Cobol* в основном ориентирован на разработку программ обработки финансовых операций, то *C* изначально задуман для системного программирования, написания компиляторов и операционных систем. Операционная система *UNIX* и большинство программ для неё созданы на *C*. Язык *C++* является результатом совершенствования и развития *C*, он обогащён технологией объектного программирования (см. ниже).

Для уровня школьников порождён легко усваиваемый *Basic* (1964 г.), а для обучения профессиональному программированию (в основном для студентов) разработан – строгий *Turbo Pascal* (1971–1990 гг.). Полагается, что *Basic* наиболее пригоден для самого первого знакомства с программированием. Однако наш опыт преподавания свидетельствует о том, что первым языком программирования, безусловно, должен быть *Turbo Pascal* или *C++*, но никак не *Basic* (имеет более трёхсот разновидностей), приучающий разработчиков приложений к *плохим* стилю и приёмам программирования.

С течением времени в ходе своего развития все языки программирования высокого уровня заимствуют друг у друга всё лучшее. Поэтому современные версии, например *Basic* и *Turbo Pascal*, по своим возможностям *вполне* могут использоваться для решения большинства научных и инженерных задач. Но вплотную подойти к богатым возможностям *C++* ещё *ни одному* языку не удалось.

Когда число строк кода программы более 10–100 тыс., то применяется специальная технология ООП – объектно-ориентированное программирование (без её использования *быстрая коллективная* работа над приложением невозможна). Эта технология подробно рассматривается в нашем пособии. Языки *C++*, *C++ANSI*, *Object Pascal* ориентированы на ООП-технологии, а *Turbo Pascal* лишь поддерживает её.

Для неспециалистов кажется *невероятным* тот факт, что, применяя языки программирования высокого уровня, приблизительно 90–95% времени, затрачиваемого на разработку программы, уходит на создание интерфейса (средств управления: окон ввода и вывода информации, пунктов меню, командных кнопок и др.). И только 5 – 10% времени посвящается решению собственно основной задачи. Для успешной борьбы с такой несправедливостью фирмы-разработчики про-

граммного обеспечения предложили специальные среды быстрой разработки приложений или *RAD-среды (Rapid Application Development)*. Такими средами являются:

- ❑ *Borland C++ Builder* (версия 5.0—выпущена в 1998 г., версия 6.0—в 2000 г., версия 2006— в 2005 г., версия 2007— в 2007 г., версия 2009— в 2008 г. (выпускается фирмой *Code gear*));
- ❑ *Borland Delphi* (версия 1.0— выпущена в 1995 г., версия 5.0—в 1999 г., версия 6.0— в 2001 г., версия 7.0—в 2002 г., версия 2005—в 2005 г., версия 2007— в 2007 г., версия 2009—в 2008 г. (выпускается фирмой *Code gear*));
- ❑ *Microsoft Visual Basic* (версия 1.0 выпущена—в 1991 г, версия 6.0—в 1998 г);
- ❑ *Microsoft Visual C++* (версия 1.0— выпущена в 1992 г., версия 6.0—в 1998 г).

При помощи большого набора специального инструментария эти среды берут на себя заботы по разработке всех удобств, облегчающих использование программ-приложений: полей ввода и вывода информации, наглядности форм её представления и многое, многое другое. При этом упомянутые среды генерируют соответствующие коды программ автоматически и практически мгновенно. В основу *RAD-среды Delphi* положен *Object Pascal*, а *RAD-сред Visual C++* и *C++ Builder* — язык *C++*. Отличия *C++ Builder 2007* от *C++ Builder 6* небольшие: появилась ещё одна стандартная библиотека и несколько визуальных компонентов, усовершенствована *Интегрированная Среда Разработки (ИСР)*. В *C++ Builder 2009* улучшена комфортность работы с базами данных, добавлены библиотека и визуальные *Internet*-компоненты, на 10% ускорен запуск *RAD-среды*. Кроме того, в *C++ Builder 2009* повышена совместимость с кодом на *Delphi* и появилась возможность применять компоненты *Delphi*. *Builder 2009* можно установить в любой из операционных систем: *Windows 2000*, *Windows XP* или *Windows Vista*. Вместе с тем, ошибки компилятора, обнаруженные нами в *Builder 5*, *Builder 6*, в *Builder 2009* не устранены (подробнее см. в п. 14.10.2).

Современный мир немыслим без *Internet* (Всемирная Глобальная Сеть *World Wide Web* была создана в 1989 г.), поэтому в настоящее время все новые среды программирования позволяют осуществлять взаимодействие приложений во всемирной сети (однако не все такие приложения являются платформенно-независимыми). Для создания *Web*-страниц используются специальные языки *разметки*, позволяющие управлять форматированием и размещением элементов страниц. Наибольшее распространение получил язык *HTML (HyperText Markup Language)* — язык гипертекстовой разметки, предложен в 1989 г.).

Язык *HTML* даёт возможность создавать документы лишь со *статическими* текстами, таблицами и изображениями. Для разработки *Web*-страниц, на которых ведётся диалог с клиентом, выполняется анимация изображений, реализуется контекстно-зависимый текстовый и графический ввод-вывод, вначале стали использоваться языки высокого уровня *Java* (разработан в 1995 г.) и *JavaScript* (разработан в 1995–1996 г.г.). Созданные с их помощью приложения являются *независимыми* как от типа используемого компьютера, так и от операционной системы. Это позволяет применять такие приложения на *всех HTTP-серверах* и *всех*

браузерах клиента. Следует отметить, что первый браузер появился в 1991 г. (с 1994 г. браузеры начала выпускать фирма *Netscape*). Язык *Java* – объектно-ориентированный язык, созданные с его помощью приложения работают на *различных* платформах (межплатформенный язык), он унаследовал синтаксис *C* и *C++*. *JavaScript* – очень похож на *Java*, но существенно проще его. *JScript* – это *JavaScript* в реализации фирмы *Microsoft*.

Фирмы-разработчики программного обеспечения, непрерывно конкурируя друг с другом, создают всё новые и новые продукты (с интервалом 1,5 – 2 года). В их приложениях не только реализуются *новые* пути решения актуальных задач, но и даётся возможность осуществлять упомянутые решения с привлечением *уже существующих* языков программирования, которые давно стали надёжным инструментарием многих разработчиков программ. Такая политика, конечно, весьма способствует продажам продуктов, повышает их востребованность.

В частности, фирма *Microsoft* предложила среду *Visual Studio.Net*, как конкурента среды программирования *Borland C++ Builder*. *Visual Studio.Net* выпускалась в различных версиях: версия 6.0 – в 2000 г., версия 2002 или 7.0 – в 2002 г., версия 2005 или 8.0 – в 2005 г., версия 2008 или 9.0 – в 2007 г. Среда программирования *Visual Studio.Net* является средством разработки *Windows*-приложений (*Windows Application*) в операционных системах *Windows-95, 98, NT, Vista* (только в *Visual Studio.Net* 2008), *Internet*-приложений (*ASP.Net*) и консольных *Dos*-приложений (*Consol Application*). Следует заметить, что консольные приложения можно разрабатывать всеми средами программирования. Основой обсуждаемой среды является *новый* компонентно-ориентированный язык программирования *C#* (читается – *си шарп* или *си диез*, создан в 1999–2000 г.г.), он *специально* разработан для технологии «*.Net*» (читается – *дот нет*). Этот самый новый язык разработки программ обеспечивает *совместную* работу компонентов, написанных с использованием *различных* языков программирования. В среде *Visual Studio.Net* можно создавать приложения при помощи следующих языков программирования: *Visual Basic* 6.0, *Visual C++*, *C#*, *JavaScript* (*JavaScript* имелся только в версии 6.0). Синтаксис *C#* похож на синтаксис *C++* и *Java*.

Платформа или оболочка «*.Net*» – это *многоязыковая* среда, она *открыта* для *свободного* включения *новых* языков (компиляторов и инструментальных средств), созданных не только *Microsoft*, но и другими фирмами. Программы, написанные с использованием технологии «*.Net*», могут функционировать на других компьютерах, *только* в том случае, когда на них установлена исполняющая (операционная) среда «*.Net Framework*» (запущен файл *dotnet.exe*), являющаяся оболочкой, надстройкой над существующими операционными системами и языками программирования. Необходимо заметить, что среда «*.Net Framework*» распространяется фирмой *Microsoft* бесплатно. Код, порождённый в среде разработки «*.Net*», носит название *управляемого* кода (*manage code*) в отличие от обычного *неуправляемого* кода (*unmanage code*). Программа с управляемым кодом *непрерывно* контролируется средой «*.Net*»: операционная среда *не позволяет* программе выполнить действия, нарушающие концепцию безопасности «*.Net*» (что повышает надёжность функционирования приложений). В настоящее время, по

прошествию около 9 лет с момента появления «.Net», только *небольшая* часть программистов (около 25%) использует эту новую весьма оригинальную технологию.

В среде разработки *Borland Developer Studio 2006* (она выпущена в 2006 г, составной частью в неё входит и *C++Builder 2006*) помимо традиционных приложений *Windows* можно создавать «.Net»-программы при помощи языков *Delphi Language* (так стал называться *Object Pascal*) и *C#*. Заметим, всё, что можно написать на *C#*, можно запрограммировать и на *Delphi Language* для «.Net» с не меньшей эффективностью [15]. Среда разработки *Borland Developer Studio 2006* (далее – *BDS*) предназначена, прежде всего, для корпоративных пользователей.

Для повышения продаж (путём существенного уменьшения цены) фирма *Borland* в 2006 г. «разрезала» *BDS* на составные части. Каждая часть включает только *один* язык и платформу. В результате появилась линейка продуктов серии *Turbo*: *Turbo Delphi* для *Win32*, *Turbo Delphi* для «.Net», *Turbo C++* для *Win32* (вариант *C++Builder*), *Turbo C#* для «.Net». Эти продукты предназначены для *небольших* групп: студентов, программистов-одиночек и индивидуальных пользователей. Каждый из семейства *Turbo* выпускается в двух редакциях: бесплатная *Explorer* (для студентов и непрофессионалов) и *Professional* (для профессионалов). Бесплатная лицензионная версия имеет не все возможности версии для профессионалов, однако её *разрешается* использовать для разработки и *коммерческих* приложений.

Наш обзор не претендует на полноту, в нём упомянуты лишь *наиболее* популярные среды разработки программ *ведущих* фирм. Поэтому отметим ещё и фирму *Sun Microsystems*, выпустившую в 2007 г. новую версию среды разработки для языков *C*, *C++* и *Fortran* под операционные системы *Solaris* и *Linux*.

Будущее покажет, насколько перспективной окажется «.Net»-технология, будет ли ей сопутствовать удача, или она погибнет (будет забыта) в результате субъективизма, инертности программистов или появления ещё более совершенной технологии.

Мода правит бал не только на подиуме, она распространяет свои неподдающиеся логическому анализу чары даже на языки программирования. К настоящему времени создано около 3 000 языков, самые распространённые из них не обязательно являются лучшими языками программирования. Начинающему выбрать «самый-самый» не представляется возможным. Даже умудрённые опытом специалисты не могут это сделать, ведь требуется в совершенстве знать каждый из них. История и фирмы-разработчики программного обеспечения, конечно, делают свой выбор, в результате популярным становится тот или иной язык. При этом следует отметить [19], что за рубежом *C++Builder* и *Delphi* такой известностью, как у нас (в постсоветском пространстве) не пользуются, там царствует *Visual Basic* (в котором даже нет пользовательских объектов...). Поэтому популярность имеет не только временную, но ещё и региональную составляющую.

Мы несколько отвлеклись от темы (место *C++Builder* среди других языков) для того, чтобы подготовить вас к следующему заключению. В настоящее время практически все задачи можно *успешно* решать при помощи целого *ряда* языков

программирования. Язык *C#* и технология «*.Net*» весьма перспективны, однако на подавляющем числе компьютеров не установлена оболочка «*.Net Framework*», что несколько препятствует их распространению. Вместе с тем далеко не всем нужны *Internet*-программы, *Windows*-приложения пока ещё *никто* не отменял, а старые платформенно-независимые механизмы основываются на *сu*-подобных языках (*Java*, *JavaScript*). Поэтому выбор для изучения *C++Builder* позволяет *уже сейчас* вооружить вас, дорогие читатели, мощнейшим инструментом разработки *Windows*-приложений, создаёт прочные основы для быстрого изучения (если вам это понадобится) любого из языков: *C#*, *Java*, *JavaScript*. Доверьтесь нашему опыту и примите во внимание рекомендацию: начинать обучение *C++* и визуальную технологию разработки приложений значительно комфортнее и проще на основе *C++Builder 6* (или *5*), но не *C++Builder 2009*. Последним продуктом и его последующими популяциями (они, без сомнения, появятся спустя 2 – 3 года) вы сможете овладеть *самостоятельно* на основе знаний, полученных из настоящего пособия.

Не для рекламы, а в качестве констатации факта заметим, что на *C++* написаны и операционная система *Windows* и многие известные прикладные программы. При этом ещё раз подчеркиваем, *C++ специально* разрабатывался для *профессиональных* программистов (системных программистов и разработчиков программного обеспечения). Поэтому для новичков в программировании он не всегда сразу понятен. Среди программистов бытует шутка: «язык *C++* был придуман только для того, чтобы программистами не становились *случайные* люди». Однако, делающие в программировании первые шаги, могут не огорчаться: среда программирования *C++ Builder* практически ликвидировала все сложности *C++*, сделала процесс разработки программ ясным и увлекательным. Да, да – очень увлекательным. Заверяем читателей: вы овладеете *основными* приёмами программирования в среде *C++ Builder*, научитесь совершенствовать ваши навыки для дальнейшего расширения набора средств по программированию задач *достаточно* сложности.

Что собой представляет пособие

Это пособие – самоучитель. Изложение ведётся в виде беседы автора с читателем, практически *все* возможности языка иллюстрируются краткими примерами, закрепляются вопросами для самоконтроля и задачами для программирования. ООП освещается *вначале* лишь в общих чертах, в объёме, необходимом для управления средой *Builder*. После овладения основными вопросами синтаксиса и семантики *C++Builder* подробно излагается технология *объектного* конструирования программ.

Цель пособия – изучить *основные* возможности языка *C++*, реализованного средами *C++Builder 6.0* и *C++Builder 5.0*. Всё изложение ведётся на базе *C++Builder 6.0*, а в тех случаях, когда эти операции выполняются по-другому в *C++Builder 5.0*, даются подробные разъяснения (но таких ситуаций менее полдесятка). Цель учебника: дать читателю такой объём глубоко осознанных знаний,

который позволил бы ему с применением справочной системы *C++Builder* или справочников по *C++* [8] и *C++Builder 6.0* [1–3, 20–21] самостоятельно решать и круг вопросов, не затронутых в пособии.

Материал самоучителя излагается так, чтобы читатель не робел перед разработкой программ. Для этого изучаемые вопросы, задачи и алгоритмы расположены по нарастанию их сложности. При этом круг тем ограничен теми, которые *наиболее* важны для программирования *подавляющего большинства* задач, возникающих при обработке и анализе данных.

Настоящее пособие не только иллюстрирует *C++Builder* примерами и задачами, подобранными по нарастанию их сложности. Прежде всего, оно является *подробным последовательным* учебником. В ходе повествования не только указываются пути решения вопросов или смысл того или иного ограничения, но и подробно разъясняется, почему эти пути должны быть именно такими, а не другими, чем обусловлено ограничение. Это способствует как пониманию материала, так и его запоминанию.

В учебнике достаточно вопросов и заданий, с которыми можно справиться, не имея перед собой компьютера, даже в транспорте. Это было сделано не для того, чтобы увеличить набор литературы для путешественников или жителей спальных районов крупных городов. Дело в том, что для глубокого понимания вопросов очень полезно «прокручивать» все шаги компьютера у себя в голове, следует уметь *читать* тексты программ. Однако каждую порцию прочитанного и, на первый взгляд, хорошо понятого материала *необходимо* закреплять его практической реализацией на компьютере. Иначе научиться программировать *невозможно*.

Основные рекомендации для читателей

Рекомендуется *последовательно* изучать урок за уроком, *отвечать* на вопросы для самоконтроля и *выполнять* задания. При этом эффективность усвоения *существенно* повысится, если полностью отлаженные программы *записывать* на отдельных листах бумаги. Именно записывать, а не распечатывать при помощи принтера, поскольку в первом случае знания надолго фиксируются в вашей памяти. Этот банк задач удобно использовать при разработке новых программ, когда требуется быстро вспомнить уже применяемый ранее алгоритм или забытую конструкцию языка.

В ходе изучения уроков вы обнаружите, что уроки, пройденные ранее, несколько подзабыты. Это вполне нормально. Невозможно всё удержать в голове. Для борьбы с этим явлением вы создадите конспект решённых задач, есть, наконец, и настоящий учебник, который разумно перелистывать не только вперёд, но и назад.

Настоятельно советуем читать коды программ-решений задач для самостоятельного программирования. Это полезно делать, когда в вашей программе что-то не клеится, и даже когда вы *самостоятельно* получили правильное решение. При этом вы много раз удостоверитесь в том, что имеется много вариантов реше-

ний. Надеемся, что *ваши* программы будут оригинальнее тех, что приведены в пособии.

Если вы, дорогой читатель, не обладаете навыками работы в операционной среде *Windows*, в текстовом редакторе *Word* и в табличном процессоре *Excel*, то смело можете *отложить* чтение настоящего пособия. Оно вам окажется более полезным лишь после овладения, пусть даже минимальными, сведениями по этим продуктам. Мы ориентируемся на читателей, которые уже умеют набирать и редактировать текст в текстовом редакторе *Word*, знают, как в различных продуктах *Windows* осуществляется выделение блока, его помещение в буфер обмена, копирование, удаление и вставка. Методы перетаскивания (буксировки), изменения размеров выделенных объектов при помощи их маркеров и мыши (метод протягивания) также полагаются известными. Впрочем, всем этим элементарным премудростям можно научиться и в среде программирования. Однако овладеть упомянутыми офисными программами вам всё равно понадобится позже при *оформлении* результатов исследований, расчётов, нобелевских докладов и прочих материалов. Поэтому не рекомендуем откладывать эту работу в долгий ящик, ведь указанными продуктами обязаны владеть инженеры, научные сотрудники, журналисты, поэты и прочая писательская братия.

Если при изучении таких языков высокого уровня, как, например, *Fortran*, *C++*, *Turbo Pascal*, *QBasic* и др., знания английского языка лишь весьма желательны, однако далеко необязательны (30 – 40 ключевых слов языка можно легко запомнить всем), то ситуация с *C++ Builder* несколько иная. Для *эффективной* работы необходимо знать английский язык хотя бы в рамках программы средней школы. Имена всех идентификаторов и команд лучше запоминаются, «с лёта» понимается их назначение, если вам известны английские слова, положенные в основу упомянутых служебных слов; информацию о них значительно быстрее в этом случае найти (и понять) в англоязычной справочной системе. Для понимания различных сообщений *RAD*-среды, информации справочной системы необходимо уметь разбираться в технических текстах на английском языке, пусть даже со словарём.

К пособию прилагается диск с программами, приведенными во всех уроках (отдельно для *C++Builder 5.0* и *C++Builder 6.0*). Наш совет: обращайтесь к ним только в исключительных случаях, программы значительно полезнее набирать и отлаживать *самостоятельно*.

В заключение желаем вам большого *интеллектуального наслаждения* в увлекательной работе по освоению *C++Builder*! Будем очень рады *любым* замечаниям и пожеланиям, касающимся данной книги.

Урок 1. Первая программа

На этом уроке вы познакомитесь с интегрированной средой разработки (*IDE – Integrated Development Environment*), а также основными определениями и этапами проектирования программ. Все эти вопросы иллюстрируются на приложении (синоним слова программа), назначение и внешний вид которого описаны ниже.

1.1. Внешний вид и назначение приложения *Умножитель*

Внешний вид приложения или его интерфейс показан на рис. 1.1. В области заголовка интерфейса находятся пиктограмма (небоскрёбы), название приложения (*Умножитель*) и стандартные граничные пиктограммы, характерные для большинства окон *Windows* (*Свернуть, Развернуть, Заккрыть*). На интерфейсе имеется также два прямоугольных *поля ввода* данных (информации), над которыми расположены надписи *Цена* и *Количество*, а также командная кнопка с заголовком *Умножение*. С использованием этого приложения имеется возможность вычислить стоимость отобранного вами товара (например, в магазине). Для этого необходимо в упомянутые поля ввода занести данные о цене и количестве товара, после чего мышкой нажать командную кнопку *Умножение*. Произведение цены товара и его количества появится в *поле вывода* под надписью *Стоимость*.

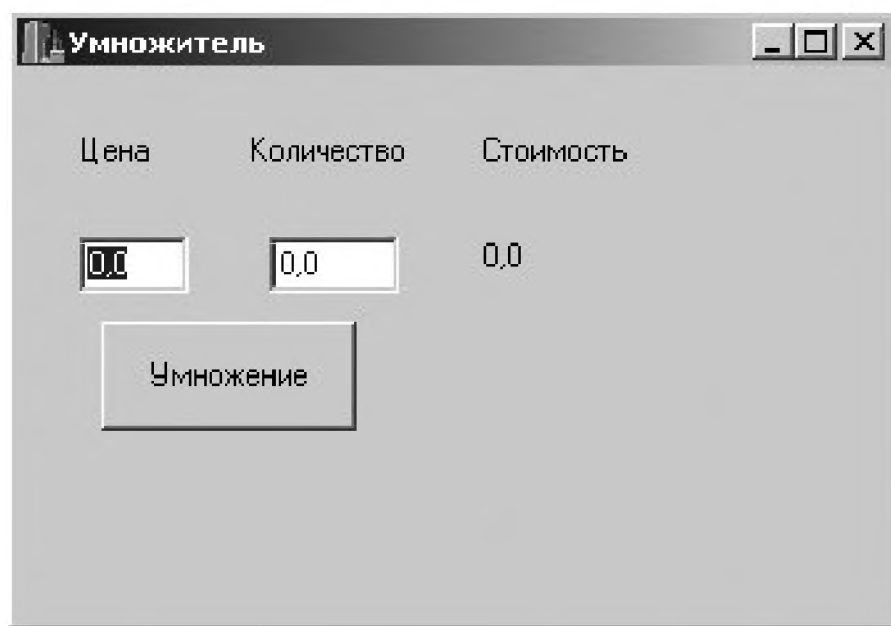


Рис. 1.1. Интерфейс программы *Умножитель*

В исходном состоянии в полях ввода и вывода установлены нулевые значения. Эта программа называется *Умножитель*, поскольку предназначена для умножения двух чисел.

Задача заключается в создании такого приложения. Она состоит из двух частей. Вначале с использованием средств визуального программирования построим интерфейс приложения, а затем разработаем программный код, обеспечивающий его функционирование. Назначение задачи состоит в том, чтобы на её основе сделать первые *глубоко осознанные* шаги в визуальном программировании и программировании на C++.

1.2. Знакомство с визуальным программированием

Когда вы первый раз увидите главное окно *IDE C++ Builder* (см. рис. 1.2), не пугайтесь! Все его вкладки, пиктограммы и меню *сразу* не понадобятся, с ними будем знакомиться *постепенно*. Далее для краткости изучаемая интегрированная среда разработки будет именоваться также *Builder*.

Интерфейс *IDE* состоит из четырёх макрокомпонентов (см. цифры на рис.1.2), их можно перетаскивать в любое место рабочего стола, изменять размеры. Познакомимся детально со всеми макрокомпонентами.

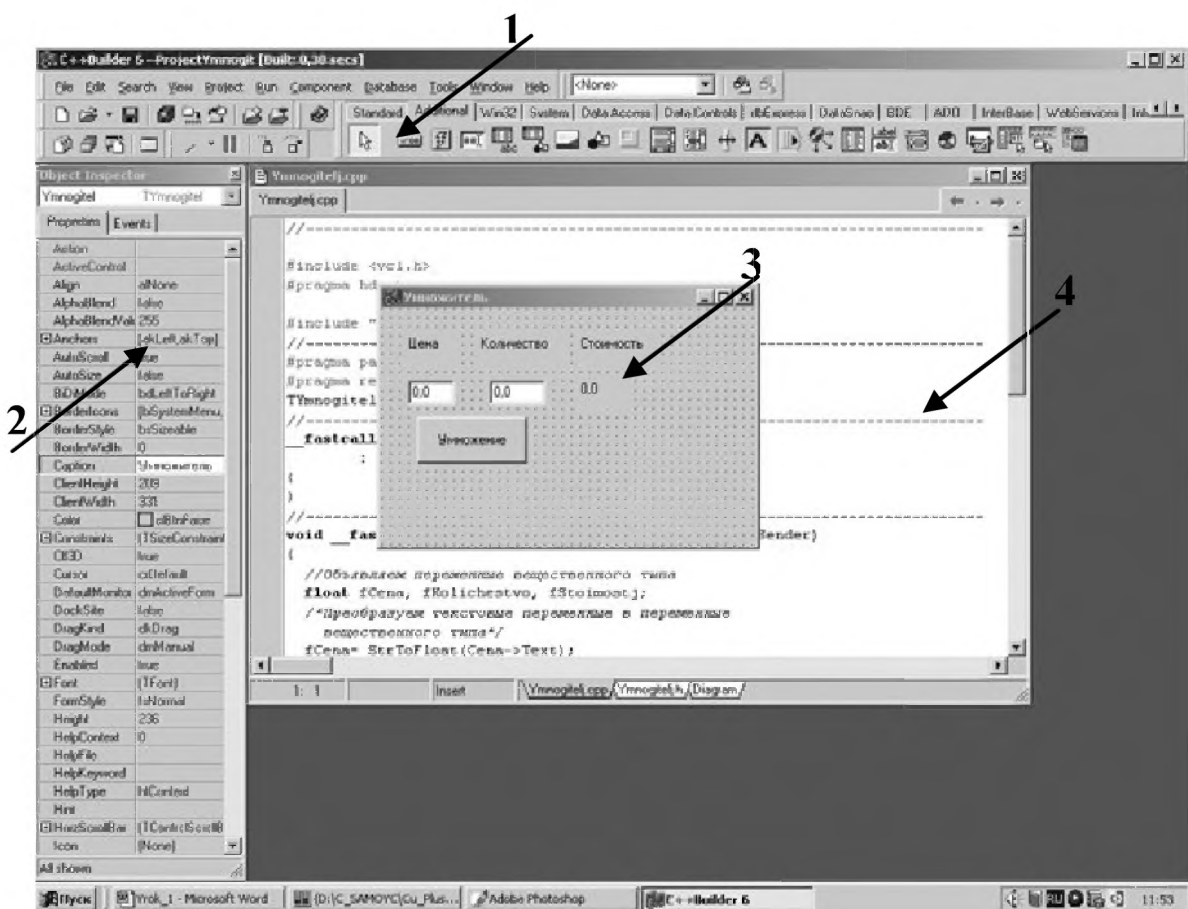


Рис. 1.2. Интерфейс C++Builder.

1– Панель Управления, 2– Инспектор Объектов,
3– Форма, Визуальный Проектировщик или Дизайнер Форм, 4– Редактор Кода

1.2.1. Форма

Дизайнер форм 3 (его ещё называют *Визуальным проектировщиком* или просто *формой*) на рис.1.2 по виду практически совпадает с интерфейсом программы *Умножитель*, такой интерфейс нам надлежит спроектировать. Однако при запуске *нового* окна приложения форма всегда пуста, на ней нет никаких элементов (см. рис. 1.3) кроме заголовка, граничных пиктограмм (*Свернуть*, *Развернуть*, *Заккрыть*) и рабочей области, равномерно покрытой точками.

Форма это прямоугольное окно, в котором разрабатывается интерфейс программы. Обычно интерфейс состоит из объектов управления программой и вывода данных. В нашем учебном приложении объектами управления являются поля ввода данных (*Цена*, *Количество*) и командная кнопка (*Умножение*). Эти объекты позволяют управлять программой. Поле вывода *Стоимость* относится к объектам вывода данных, оно предназначено для показа результата вычислений. Надписи-заголовки *Цена*, *Количество* и *Стоимость* представлены на интерфейсе также при помощи полей вывода, они только информируют пользователя о значении данных в полях, над которыми расположены.

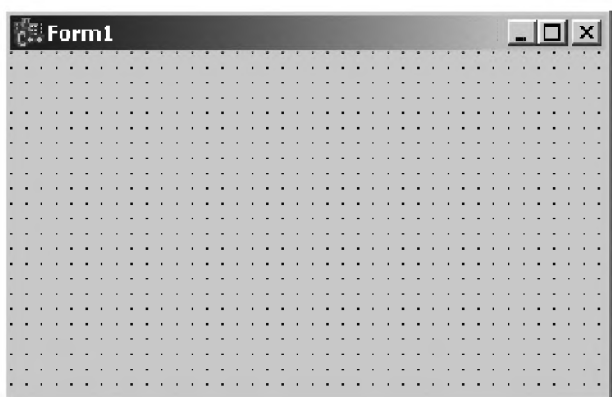


Рис. 1.3. Визуальный проектировщик

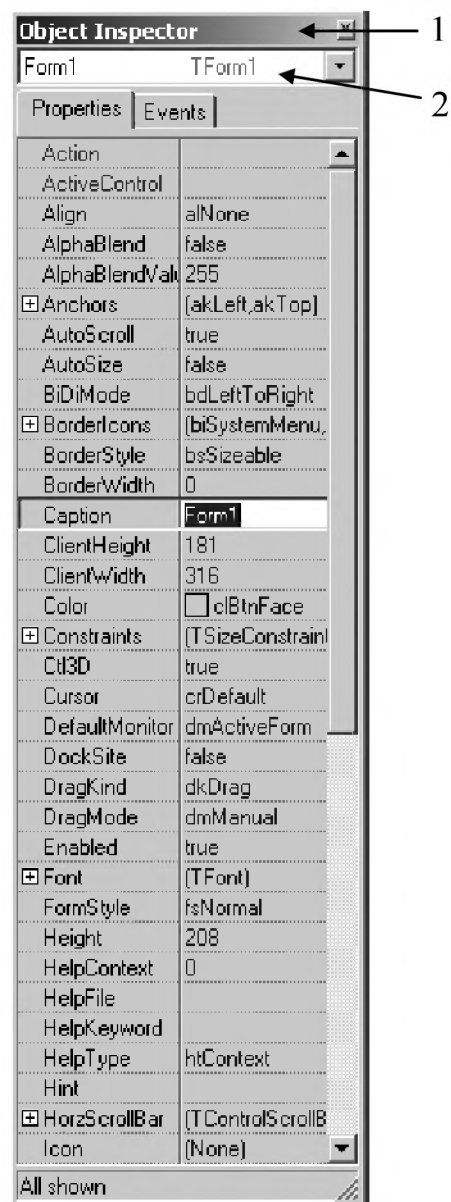


Рис. 1.4. Инспектор Объектов.

1 – заголовок; 2 – раскрывающийся список, состоит из объектов формы и самой формы

Местоположение и габариты формы удобно изменять мышью. Однако эти и многие другие параметры (или свойства) формы можно устанавливать при помощи *Инспектора Объектов* (см. 2 на рис. 1.2). Проиллюстрируем эту возможность вначале на свойстве *Caption* (заголовок).

Форма, при своём автоматическом первом появлении (инициализации) имеет заголовок по умолчанию *Form1* (см. рис. 1.3). Инспектор объектов в этом случае выглядит так, как показано на рис. 1.4. Он состоит из заголовка (1), раскрывающегося списка из объектов формы и самой формы (2) и двух вкладок (закладок, страниц) *Properties* (*Свойства*) и *Events* (*События*). По умолчанию активна вкладка *Свойства*. Каждая вкладка состоит из двух столбцов. В первом столбце в алфавитном порядке перечислены свойства (или события) выбранного объекта формы или самой формы, а во втором столбце находятся поля ввода значений свойств (или имён функций по обработке событий).

При последовательном размещении на форме упомянутых выше объектов интерфейса, в такой же последовательности в раскрывающемся списке будут появляться и названия этих объектов с их типом. В раскрывающемся списке указывается имя выбранного (выделенного) на форме объекта. Объект выделяется щёлчком мыши или выбором в этом списке.

Изменим заголовок формы *Form1* на *Умножитель*. Для этого щелчком мыши по *Инспектору Объектов* сделаем его активным. Можно для этой цели применить клавишу *F11*, последовательное нажатие которой делает активными поочередно *Форму*, *Редактор Кода* и *Инспектор Объектов*. Далее выберите свойство *Caption*, в поле его ввода наберите слово *Умножитель*. Ход набора этого слова сразу отражается (практически без задержки) в заголовке формы.

Свойством *Caption* обладают те объекты интерфейса, которые предназначены для вывода информации. Для формы это свойство определяет её заголовок. Заголовок можно набирать как латиницей, так и буквами любого национального алфавита (например, кириллицей). Заголовок формы является заголовком интерфейса будущей программы.

Ширину и высоту формы помимо мыши можно также переустановить при помощи соответственно свойств *Width* и *Height*, набрав в их полях ввода значения габаритов формы (измеряются в пикселях). Если требуемое свойство не видно в списке свойств, то примените полосу прокрутки.

Свойство *Align* (выравнивание) позволяет запрограммировать местоположение интерфейса справа (*alRight*), слева (*alLeft*), внизу (*alBottom*) или вверху (*alTop*) экрана, либо развернуть его на весь экран (*alClient*).

Мы не станем сейчас рассматривать назначения всех свойств, их достаточно много и каждое из них по-своему важно. Например, с использованием свойства *Color* имеется возможность подобрать желаемый цвет формы.

1.2.2. Тип объекта

Каждый объект формы и сама форма *должны* иметь только *уникальные* (не повторяющиеся в программе) имена, которые присваиваются в *Инспекторе*

Объектов в свойстве *Name*. Имя формы по умолчанию – *Form1* (совпадает с заголовком по умолчанию). Поэтому в поле, расположенном под заголовком окна *Инспектора Объектов* (см. рис. 1.4), вы увидите *Form1:TForm1*. После двоеточия здесь указан тип (*Type*) *TForm1* формы с именем *Form1*. В поле ввода свойства *Name* наберите *Ymnogitelj*, после чего вместо *Form1:TForm1* появится *Ymnogitelj:TYmnogitelj*. Это имя и имена всех других объектов формы программы (о них будет рассказано ниже) необходимо набирать *только* латинскими буквами. Буква «*T*» в названии типа неизменна, а вот следующая за ним часть названия (без пробела) изменяется автоматически при изменении имени формы.

Что же такое тип? Понятие *тип* в программировании является одним из ключевых. Тип имеют не только объекты интерфейса (форма, поля ввода и вывода, командные кнопки, полосы прокрутки, радиокнопки и др.), но и константы, переменные и другие «члены семьи» любого языка программирования.

Приведём примеры характеристик отдельных объектов интерфейса. *Командная кнопка* имеет определённые габаритные размеры, местоположение на интерфейсе, цвет, служит для ввода *команд*. *Поле ввода* данных также имеет свои габариты, расположение, цвет, однако в отличие от командной кнопки оно обычно предназначено для ввода *данных*. Таким образом, у разнородных объектов могут быть одинаковые свойства, например: высота, ширина, местоположение, цвет и др. Однако у объектов имеются и характерные особенности, которые отличают один тип объекта от другого, например, командную кнопку от поля ввода.

Рассмотрим несколько отвлечённый пример. Любого человека можно описать набором параметров или свойств: имя, год рождения, место жительства, рост, вес, цвет глаз и волос и пр. Люди могут приобретать знания, обладают речью, способны изучить *C++Builder*. Набор таких *формальных* характеристик для *каждого* человека наполняется *конкретным* (фактическим) содержанием (заданный рост, вес и др.).

Животного, например медведя, можно характеризовать почти таким же набором параметров. Вот только человек Миша способен, например, изучить *C++Builder*, а медведь Миша – нет. Поэтому, если у нас имеется объект *Misha*, то тип такого объекта *TChelovek* или *TMedvedj* полностью разрешает вопрос о том что это за объект. В программировании ответ на этот вопрос очень важен, например, для определения объёма памяти, который необходим для размещения объекта указанного типа в памяти компьютера.

Весь набор свойств (и событий) конкретного визуального объекта формы и самой формы определяет его возможности и называется типом или классом.

1.2.3. Правильный идентификатор

Все имена или *идентификаторы* объектов программы (имена объектов, констант, переменных, функций, типов, структур, классов и др.) набираются без пробелов, могут содержать арабские цифры, латинские прописные или строчные буквы и знак подчёркивания. Однако начинаться они *должны* с буквы, а не с цифры. Знак подчёркивания в начале идентификатора применять не рекомендуется

(чтобы случайно не заблокировать некоторые стандартные функции и переменные, которые также начинаются с этого знака). Строчные и прописные буквы в идентификаторах *различаются*, поэтому *Ymnogitelj*, *ymnogitelj* и *YMNOGITELJ* – три разных имени. Длина идентификатора *не ограничивается*, однако *Builder* распознаёт только *первые* 250 символов (т. е. значимы, принимаются им во внимание не более 250 символов). Идентификаторы, удовлетворяющие перечисленным требованиям, называются *правильными идентификаторами*.

Мы не рекомендуем использовать в идентификаторах более 8 – 11 символов, поскольку более длинные имена затрудняют беглое чтение программы. Языки высокого уровня придуманы для того, чтобы программисту можно было не запоминать номера команд и адреса ячеек, где хранятся используемые переменные. Поэтому очень неразумно поступают те разработчики программ, которые своим идентификаторам присваивают ничего не значащие, легко забываемые имена или даже отдельные буквы. Идентификаторы могут в лаконичной форме ясно *подсказывать* программисту суть вычисляемой задачи (если идентификатор – имя программы), сущность констант, переменных (и их тип) и других объектов программы. Сформулируем один из *основных* пунктов хорошего стиля программирования: необходимо применять информативные, «умные» идентификаторы, способствующие лёгкому пониманию (чтению) текста программы.

1.2.4. Поле ввода

Расположим теперь на форме *Поле ввода* (или окно ввода) для набора значения цены товара. Для этого нам понадобится ещё один компонент *IDE*, называемый *Палитрой Компонентов*. *Палитра Компонентов* расположена на макрокомпоненте *Панель Управления* (см. 1 на рис. 1.2 и рис. 1.5), отдельно показана на рис. 1.6, она содержит большой набор различных заготовок-компонентов, используемых для сборки интерфейса *любой* программы. Каждый компонент на палитре представлен отдельной пиктограммой. Все они систематизированы на многочисленных вкладках (для их просмотра воспользуйтесь полосой прокрутки). В *Builder 5.0* и *Builder 6.0* имеется соответственно 19 и 29 вкладок. При зависании курсора мыши над каждым компонентом (кроме первого, он является общим для всех вкладок) появляется всплывающая подсказка о его названии.

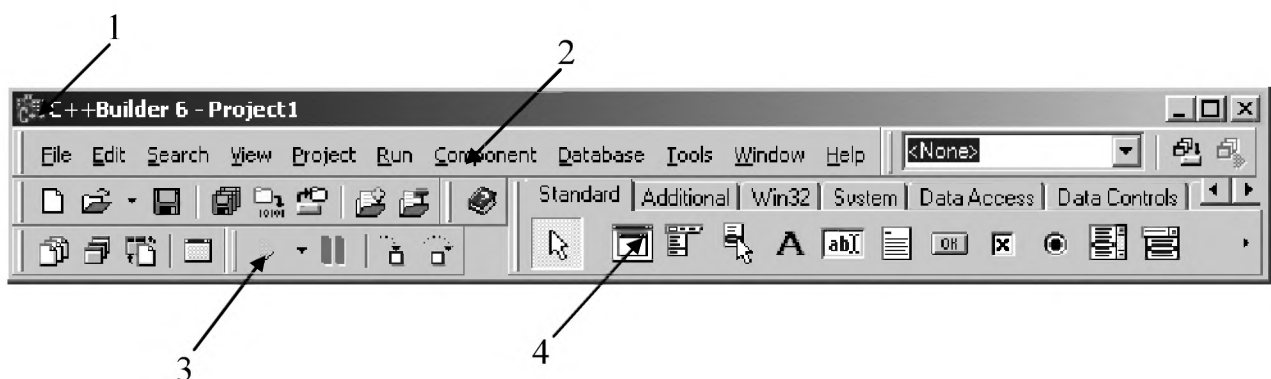


Рис. 1.5. Панель Управления.

1 – заголовок, 2 – горизонтальное (или главное) меню,
3 – кнопка запуска программы, 4 – панель Палитра компонентов

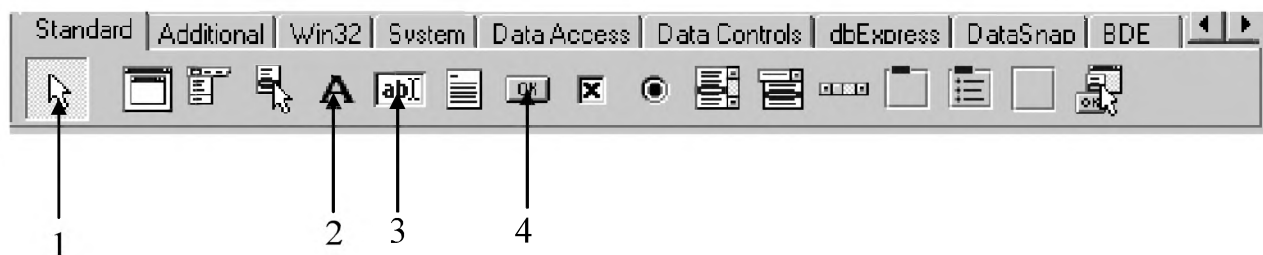


Рис. 1.6. Палитра Компонентов.

1 – Выделить/Снять выделение; 2 – компонент Label (поле вывода);
3 – компонент Edit (поле ввода); 4 – компонент Button (командная кнопка)

Для размещения на форме поля ввода информации вначале на вкладке *Standard* Палитры Компонентов выберите компонент *Edit* (см. 3 на рис. 1.6), далее также именуемый *Поле ввода*. После этого щёлкните указателем мыши приблизительно в той части формы, где требуется поместить проектируемое поле ввода цены товара (см. рис. 1.1). Появится *Поле ввода* с маркерами. Придайте этому полю желаемые габариты и отбуксировать в требуемое место на форме. *Поле ввода*, расположенное на форме, называется *объектом*, а не компонентом. Ещё раз подчеркнём особенность терминологии: на панели компонентов *Поле ввода* называется *компонентом*, а на форме – *объектом*. Такая трансформация происходит и для всех других компонентов *Панели Компонентов*.

Имеется и другой способ размещения объекта (этого или любого другого) на форме: сделайте *двойной* щелчок по пиктограмме компонента на *Палитре Компонентов*, в результате объект (с именем по умолчанию) появится *в центре* формы и его можно отбуксировать в более подходящее место, придать необходимые габариты. Уничтожить *выделенный* объект (с чёрными квадратными маркерами по сторонам) на форме можно при помощи клавиши *Delete*.

После появления поля ввода в нём имеется текст *Edit1*, совпадающий с именем по умолчанию (в свойстве *Name*). Создайте на форме ещё одно поле ввода (для ввода количества товара), оно получит имя по умолчанию *Edit2*. Последующие поля будут именоваться *Edit3*, *Edit4* и т. д. При попытке именовать два поля одинаково, например, *Edit2*, *Builder* выдаст предупреждение о допущенной ошибке: *Компонент с именем Edit2 уже существует* (см. рис. 1.7).

Имена объектов формы должны быть уникальными (не должны повторяться) правильными идентификаторами (см. предшествующий пункт), даже если они разного типа.



Рис. 1.7. Сообщение о допущенной ошибке

Вернёмся к первому полю ввода. Выделите это поле, затем указателем мыши (или клавишей *F11*) перейдите в *Инспектор Объектов* на вкладку *Свойства*, найдите там и активизируйте свойство *Text* (Текст). В поле ввода этого свойства (находится справа от имени свойства) вместо значения *Edit1* введите 0,00 (исходное значение поля).

Поскольку создано поле для ввода *цены* товара, то разумно автоматически сгенерированное, ничего не значащее имя *Edit1* заменить (в свойстве *Name*), например, на имя *Cena* (эта операция, не является обязательной, однако желательна для получения понятной программы).

Поле с именем *Edit2*, предназначенное для ввода количества товара, поместите справа от поля *Cena*, присвойте ему новое имя *Kolichestvo*, а в само поле (свойство *Text*) запишите исходное значение «0,00» (см. рис. 1.1).

Таким образом, мы создали два экземпляра объекта типа *TEdit*. Каждый из них (*Cena*, *Kolichestvo*), характеризуется *своими* именем и местоположением, а их габариты пусть будут одинаковыми. При необходимости можно создать *произвольное* число экземпляров такого типа.

1.2.5. Поле вывода

Поле вывода (иногда его именуют полем надписи или окном вывода) используется для *вывода* информации. В *Builder* такой объект называется *Label* (метка), поскольку он часто применяется для того, чтобы *позначить* различные объекты интерфейса (вывести их названия). Пиктограмму компонента *Label* вы легко найдёте на закладке *Standard Палитры Компонентов* (см. 2 на рис. 1.6). Для вывода информации существуют и другие средства, с которыми познакомимся на других уроках.

Содержимое *Поля вывода* может иметь различный характер. В разрабатываемом приложении в это поле (свойство *Caption*) с именем *Stoimostj* (свойство *Name*) помещается конечный результат вычислений (при помощи программного кода). В этом случае значение свойства *Caption* изменяется в зависимости от введенных значений в полях *Cena* и *Kolichestvo*. Начальное значение этого свойства объекта *Stoimostj* установите равным 0,00.

Когда вы сотрёте исходное значение (*Label1*) свойства *Caption* объекта *Stoimostj*, его горизонтальный размер уменьшится до минимально возможного значения (объект сожмётся в узкую вертикальную полоску). Если вы введёте в поле ввода *Caption* упомянутое исходное значение «0,00», то поле вывода расширится на ширину трёх цифр и запятой – необходимого места для вывода введенного числа. Такой режим работы объекта обеспечивается свойством *AutoSize* (автоматическая подстройка размера объекта под его содержимое), которое является активным по умолчанию.

Иногда функционирование рассматриваемого свойства оказывается полезным. В нашей программе использовать его не будем. Для отключения свойства *AutoSize* щёлкните по нему мышью, затем – по кнопке раскрывающегося списка, далее исходное значение *true* (истина) измените на *false* (ложь). Значение, кото-

рое вы изменили, имеет логический или булевский тип (*bool*). Для свойства *AutoSize* можно установить только одно из двух значений *true* либо *false*.

При помощи объектов типа *TLabel* расположите на форме заголовки-подсказки: *Цена*, *Количество*, *Стоимость* (см. рис. 1.1). Присвойте им, например, имена: *Zagolovok1*, *Zagolovok2*, *Zagolovok3*. В программе они используются лишь для идентификации объектов–надписей, поэтому им можно оставить и исходные значения (значения по умолчанию). В этих объектах значения в свойстве *Caption* изменяться не будут, останутся постоянными в ходе всей работы программы. Если у вас всё получилось так, как показано на рис. 1.1, то переходите к размещению на форме последнего объекта интерфейса.

1.2.6. Командная кнопка

Командные кнопки служат для ввода команд. В нашем приложении при нажатии такой кнопки с заголовком *Умножение* перемножаются данные, введенные в полях ввода *Цена* и *Количество*. На рис. 1.6 видно как выглядит компонент-заготовка кнопки, он расположен на вкладке *Standard Палитры Компонентов*, в *Builder* он именуется как компонент *Button* (кнопка). Объект типа *TButton* с именем *Ymnogenie* (свойство *Name*) и заголовком *Умножение* (свойство *Caption*) разместите на форме как показано на рис. 1.1.

Имя кнопки *Ymnogenie* и её заголовок *Умножение* могут быть другими, также как могут быть иными габаритные размеры и местоположение командной кнопки. Своими действиями вы породили лишь конкретный экземпляр объекта типа *TButton*.

Вы закончили создание интерфейса, при этом не написали *самостоятельно* ни строчки кода приложения. За вас эту работу выполнял *Builder*. Вами создан целый ряд объектов, два из них имеют тип *TEdit*, четыре объекта принадлежат типу *TLabel* и по одному *TYmnogitelj* и *TButton*. Все объекты обязательно обладают свойством *Name*.

1.3. Разработка кода программы

1.3.1. Что такое алгоритм

Для объяснения происхождения слова *алгоритм* перенесёмся в девятый век, в древний город Багдад. В то время он был ещё совсем юным, поскольку основали его только в восьмом веке вблизи разрушенного Вавилона. Появился он для того, чтобы стать столицей арабского халифата – огромного государства. В его состав вошли кроме арабских государств, целиком помещавшихся на Аравийском полуострове, также Палестина, Сирия, Месопотамия, Персия, Закавказье, Средняя Азия, Северная Индия, Египет, Северная Африка и Пиренейский полуостров. Багдад сделался центром арабской культуры, которая развивалась *на основах* воспринятой арабами культуры таджиков, хорезмийцев, азербайджанцев, египтян, персов и народов древней Греции и Индии. Поскольку многие из представи-

телей народов, вошедших в халифат, писали на арабском языке, то впоследствии работы мудрецов этих народов были несправедливо включены в число достижений арабов. Арабы *только* завоевали эти народы, да дали возможность их умельцам и мудрецам плодотворно работать, собрав их в едином центре (создали прототип Академгородка).

Так вот, среди местных мудрецов звездой первой величины был «невыездной» великий узбекский (хорезмийский) математик и астроном Мухамед бен Муса аль-Хорезми, уроженец города Хорезма (Хивы). Наиболее ценными для развития математики являются труды аль-Хорезми по алгебре и арифметике. Благодаря аль-Хорезми европейцы, в частности, познакомились с индийскими методами записи чисел: с употреблением нуля, с разрядностью цифр числа. С лёгкой руки европейцев эти цифры и по сей день несправедливо именуются *арабскими*. Механизм этого казуса понятен: европейцы почерпнули эти сведения из книги, автор которой жил в арабском государстве и писал на арабском языке.

Имя аль-Хорезми оказалось связанным с термином *алгоритм* совершенно случайно. Дело в том, что латинизированное имя аль-Хорезми превратилось в *алгоритмус*, а затем уж в *алгоритм*. Под этим термином стали иметь в виду нумерацию пунктов, выполняемую индийскими числами, а впоследствии им начали выражать всякую систему или последовательность вычислений или действий.

Сейчас, дорогой читатель, вы вполне подготовлены правильно воспринять и навсегда запомнить следующую формулировку: последовательность операций, выполняемую при решении задачи, называют методом ее решения или *алгоритмом*.

1.3.2. Алгоритм программы

Вернёмся к нашей программе. После ручного набора вещественных чисел в полях ввода *Цена* и *Количество* эти числа в *текстовом* виде (последовательность символов) *хранятся* в свойствах *Text* объектов формы с именами соответственно *Cena* и *Kolichestvo*. В объекте *Stoimostj* результат вычисления (произведение цены и количества товара) записывается в свойство *Caption* в текстовом (или строковом) виде.

Builder самостоятельно генерирует заготовку функции (фрагмент кода программы) по обработке нажатия клавиши *Умножение*. Способы порождения этой заготовки будут рассмотрены ниже. Наша задача состоит лишь в том чтобы, согласно синтаксису *Builder*, вписать в упомянутую функцию необходимый код. Перечислим операции, выполняемые этим кодом.

1. Преобразовать текстовое значение свойства *Text* объекта *Cena* к вещественному типу, затем это преобразованное значение запомнить в первой переменной вещественного типа.
2. Преобразовать текстовое значение свойства *Text* объекта *Kolichestvo* к вещественному типу, после чего это преобразованное значение записать во второй переменной вещественного типа.
3. Вычислить стоимость покупки: получить произведение упомянутых

выше переменных вещественного типа. Запомнить полученное значение в третьей переменной вещественного типа.

- Преобразовать значение третьей вещественной переменной к текстовому типу и вывести в свойстве *Caption* объекта *Stoimostj*.

Перечисленные операции являются алгоритмом разрабатываемой подпрограммы. Словесные формулировки его шагов переведем на язык понятный *Builder*. Этому посвящён следующий пункт пособия.

1.3.3. Способы порождение заготовки функции

Укажем способы порождения заготовки функции по обработке события, связанного с нажатием командной кнопки:

- ❑ *Дважды щёлкнуть* по командной кнопке *Умножение*.
- ❑ Выделить командную кнопку *Умножение*. После этого перейти в *Инспектор Объектов* и сделать активной вкладку *События*, найти событие *OnClick* (при нажатии). Далее осуществить *двойной щелчок* в поле ввода этого события (находится справа от имени события). После этого в поле ввода появится имя функции *YmnogenieClick*.

Отметим, что в первом способе в упомянутом поле ввода имя функции *YmnogenieClick* появится автоматически. Имя этой функции состоит из имени кнопки *Ymnogenie* и названия события *Click*.

При использовании любого из способов вы окажетесь в окне *Редактора Кода* (или *Редактора текста программы*), предназначенного для набора и редактирования текста программы. В этом окне находится заголовок и пустое тело функции:

```
void __fastcall TYmnogitel::YmnogenieClick(TObject *Sender)
{
}
```

Тело функции – это код внутри фигурных скобок. Пока в этих скобках никакого кода нет, там лишь приветственно мигает текстовый курсор. Вновь перенестись в *Дизайнер Форм* можно при помощи указателя мыши или клавиши *F11*, которая поочерёдно делает активными все макрокомпоненты. При написании текста программы удобно также для достижения этой цели использовать клавишу *F12*, она осуществляет переключение с любого макрокомпонента *Builder* в *Дизайнер Форм* или *Редактор Кода*. Последующие её нажатия коммутируют между собой *Дизайнер Форм* и *Редактор Кода*.

1.3.4. Заголовок функции

Не огорчайтесь непонятности автоматически созданного текста, который вы видите в окне *Редактор Кода*. Со временем ситуация полностью прояснится. Сейчас же бегло расскажем о назначении элементов *заголовка* функции, расположенного над открывающей фигурной скобкой. *TYmnogitel* – это тип или класс формы объекта *Ymnogitel*. *YmnogenieClick* – имя функции, которая обрабатывает

событие (*OnClick*), порождаемое при нажатии кнопки *Ymnogenie*. В классе *TYmnogitel* (его описание находится в отдельном файле) приводится заголовок (или прототип) функции с именем *YmnogenieClick*. Сама же функция описана в файле реализации с расширением «.cpp», который вы видите перед собой в окне *Редактор Кода*.

Функции, объявленные в описании класса, называются *методами* класса. Их ещё именуют *компонентными функциями* или *функциями-элементами, функциями-членами* (существуют и другие названия). Далее будем использовать первый вариант названия.

Объявление (прототип) и описание метода можно привести в одном файле. Однако так поступать *очень уж неразумно*. Поскольку этим перечёркивается одно из основных достоинств *C++* в сравнении с другими языками программирования высокого уровня. Используя два *различных* файла, вы разбиваете программу на логически *самостоятельные* части, их называют модулями. При этом улучшается ясность программы, облегчается возможность её отладки, сопровождения и модернизации.

Builder знает, что функция *YmnogenieClick* – это реализация прототипа, описанного в классе *TYmnogitel*, поскольку впереди её имени стоит имя класса с двумя двоеточиями «*TYmnogitel::*». Знак «*::*» является операцией, которая называется по-разному: *уточнением доступа к области видимости (действия)*, расширением области *видимости (действия)*, *указанием области видимости*. Таким образом, «*TYmnogitel::*» означает, что функция *YmnogenieClick* объявлена в классе *TYmnogitel*.

Математические функции всегда возвращают какое-то значение. Например, функция *cos (x)* возвращает вещественное значение, значение косинуса своего аргумента *x*. Вместе с тем в *Builder* имеются функции, которые выполняют действия процедурного характера, например, выводят сообщение, производят очистку экрана компьютера и многое другое. Для таких функций в *C++* имеется тип данных, называемый *void* (пустота, пустой). Этим служебным словом начинается обсуждаемая нами строка, она является заголовком функции *YmnogenieClick*, которая принадлежит классу *TYmnogitel* и возвращает значение типа *void*. Если бы эта функция возвращала число вещественного типа (в *C++* он называется *float*), то в заголовке её описания вместо *void* стояло бы слово *float* (плавать, т. е. число с плавающей точкой).

После имени функции в круглых скобках приведено описание единственного аргумента: (*TObject *Sender*). *TObject* – это тип аргумента, а **Sender* – аргумент-указатель (отправитель, от *to send* – посылать) на объект, который отправил сообщение. В нашей функции этот аргумент использоваться не будет.

__fastcall – это *директива* или опция *компилятора* (о компиляторе рассказывается в п. 1.4.2). Когда она поставлена в заголовке метода, то это влияет на процесс компиляции и обеспечивает передачу аргументов (параметров) функции в быстрые регистры, что ускоряет вызов функции. Для методов класса эту опцию следует указывать *всегда*. Однако для иных функций опцию *__fastcall* устанавливать целесообразно, но *не обязательно*.

1.3.5. Задание переменных

Словесный вариант алгоритма (см. п. 1.3.2.) преобразуем в команды *Builder* и впишем их между фигурными скобками, расположенными под заголовком функции *YmnogenieClick*. Эти скобки часто называют *операторными скобками*, они указывают начало и конец логического блока, применяются не только для обозначения тела функции. Согласно алгоритму в нашей функции должны иметься три переменные вещественного (или действительного) типа (напоминаем, он называется *float*) для хранения значений цены товара, количества купленной продукции и суммарной стоимости покупок.

Имена этих переменных должны быть уникальными информативными правильными идентификаторами. Как отмечалось выше, информативность идентификаторов переменных (и всех других объектов программы) ускоряет разработку программы, делает её легко понятной. Для *Builder* имена переменных «*a*», «*b*» и «*c*» (или «*a1*», «*a2*», «*a3*») вполне допустимы. Однако человеку более ясными будут, например, имена, близкие к словам *Cena*, *Kolichestvo* и *Stoimostj*. Эти идентификаторы уже используются в качестве имён объектов интерфейса (*новаторное* их применение недопустимо), поэтому для упомянутых переменных можно, например, выбрать имена: *fCena*, *fKolichestvo* и *fStjimostj*. Буква *f* подсказывает, что указанные переменные являются переменными вещественного типа (первая буква от слова *float*). Такие префиксы в идентификаторах могут состоять как из одной, так и нескольких букв имени используемого типа переменных. Упомянутый способ именования переменных применял программист фирмы *Microsoft* венгр по национальности Чарльз Шимоньи (венг. *Karoly Simonyi*) ещё при разработках первых версий операционной систем *MS-DOS*. Когда коллеги спрашивали у него, почему его идентификаторы такие непонятные, он в шутку отвечал, мол, это потому, что они написаны на *венгерском* языке. Так ли было дело на самом деле или несколько иначе, мы не знаем, однако в настоящее время этот способ называется *способом венгерской нотации*. Он широко применяется профессиональными программистами.

Переменная типа *float* в оперативной памяти компьютера занимает 4 байта. Для выделения трёх ячеек памяти (с объёмом в 4 байта каждая) с целью размещения в них переменных *fCena*, *fKolichestvo* и *fStjimostj* необходимо за открывающей фигурной скобкой (после заголовка функции) записать такую строку:

```
float fCena, fKolichestvo, fStoimostj;
```

Таким образом, необходимо вначале указать тип, после пробела (одного или более) через запятую следует перечислить все переменные этого типа в разрабатываемом логическом блоке. После запятой, согласно требованиям оформления *любого* текста, *желательно* поставить пробел. Большее число пробелов после запятой, также как и пробелы перед запятой, синтаксисом языка программирования допускаются, однако неоправданно увеличивает длину строки, противоречат правилам оформления текста. Такое определение *обязательно* должно завершаться точкой с запятой. Процесс выделения места в памяти компьютера для

размещения переменных называется *объявлением* (или *заданием*, *описанием*) *переменных*.

В общем случае формат объявления переменных *одинакового* типа выглядит так:

```
Tip_Peremennoj Perem_1, Perem_2,..., Perem_n;
```

Можно каждую переменную располагать на отдельной строке:

```
float fCena, //Цена товара
      fKolichestvo, // Количество купленного товара
      fStoimostj; //Стоимость купленного товара
```

Разрешена и такая форма задания переменных:

```
float fCena; //Цена товара
float fKolichestvo; // Количество купленного товара
float fStoimostj; //Стоимость купленного товара
```

1.3.6. Однострочный комментарий

Текст после двух наклонных чёрточек (правых слешей) называется *комментарием*. Для работы программы он *совершенно не требуется* и полностью *игнорируется* при компиляции. Он нужен лишь для ясности текста программы, важен при её разработке, изучении и сопровождении. Его необходимо вписывать *по ходу* разработки программы (при определении тех или иных её объектов или блоков), а не тогда, когда программа уже отлажена. В последнем случае дело до комментариев, как правило, не доходит. Комментарием считается всё, что следует после наклонных косых вплоть до конца строки. Комментирование программы, вводимых в ней объектов, блоков и функций является одним из *очень важных* пунктов хорошего стиля программирования.

В нашем случае такие подробные комментарии, скорее всего, излишни (они показаны только в качестве иллюстрации комментариев), ведь мы использовали очень информативные идентификаторы, которые просто «кричат» каждому о том, что они собой представляют и какого они типа. Поэтому фрагмент кода функции, отвечающий за *описание* переменных лучше записать так:

```
//Объявляем переменные вещественного типа
float fCena, fKolichestvo, fStoimostj;
```

1.3.7. Инициализация переменных

При объявлении *любой* переменной ей можно назначить или присвоить определённое допустимое *стартовое* (начальное) *значение*. Например:

```
float fCena= 2.70, fKolichestvo= 3.5,
      fStoimostj= fCena* fKolichestvo;
```

Иногда такая возможность бывает полезной. В этом случае переменные будут не только объявлены, но и *определены* (или *инициализированы*). Это означает, что под переменную вначале выделяется место определённого объёма, а затем в этот участок памяти *записывается* конкретное стартовое допустимое значение.

В нашем случае допустимым значением является значение вещественного типа. Здесь необходимо заметить, что дробная часть числа отделяется от целой не запятой, а точкой; знак умножения в математических выражениях обозначается звёздочкой. Имя типа *float* напоминает о том, что в значениях переменных имеется точка, которая может по воле программиста перемещаться (плавать) по числу, меняя его форму записи. В значениях переменных других вещественных типов (о них рассказывается на третьем уроке) также имеется «плавающая» точка, являющаяся «водоразделом» между целой и дробной частями числа. Плавающая точка позволяет *любое* вещественное число записать по-разному, например, так:

```
2002.08  
2.00208e3  
200208e-2
```

В этих представлениях $e3=10^3$, $e-2=10^{-2}$. Тип *float* включает в себя не все вещественные числа, а только те, которые находятся в интервале значений от $3,4 \cdot 10^{-38}$ до $3,4 \cdot 10^{38}$. Они могут быть как положительными, так и отрицательными. Регистр латинской буквы *e* (при её наборе) не имеет значения.

1.3.8. Конфигурация компьютера

У внимательного читателя, видимо, возник вопрос о том, почему в поля ввода *Цена* и *Количество* вводились стартовые значения «0,00», в которых дробная часть числа отделялась от целой части не точкой, а запятой? Всё дело в том, что в бывшем Советском Союзе, а теперь и во всех странах СНГ разделение целой и дробной частей числа осуществляется при помощи запятой. В связи с этим в русифицированной операционной системе *Windows* (и её приложениях, например, *Word* или *Excel*) и для русскоязычных потребителей при программировании интерфейса приложений используется разделительная запятая. В противном случае разделительная точка воспримется, как ошибка ввода. О способах *обработки* таких ошибок расскажем на втором и третьем уроках, сейчас же сообщим, как поступать в том случае, когда разрабатываемая вами программа предназначена для стран запада или востока, в общем, для тех, кто пользуется разделительной точкой, а не запятой.

Вам необходимо найти системный файл *Windows* с именем *win.ini* (обычно он находится в подкаталоге *C:\ Windows*), зайти в него (например в *FAR* при помощи клавиши *F4*), затем в разделе *[intl]* найти строку *sDecimal* и в ней запятую заменить точкой, далее запомнить изменения в файле (нажать клавишу *F2*) и перезагрузить компьютер. Напоминаем, что в тексте программ *всегда* используется *только* разделительная точка.

1.3.9. Оператор присваивания

Все выражения в п. 1.3.7, которые через запятую перечислены после указания типа *float*, называются *операторами присваивания*. Знак «=» называется *знаком присваивания*. Он оператор присваивания разбивает на две части: правую и левую. В правой части в общем случае находится математическое выражение, зна-

чение которого необходимо вычислить (например, $fCena * fKolichestvo$), а в левой части – имя той ячейки в памяти компьютера, куда это значение следует поместить (запомнить или *присвоить*). Ясно, что в правой части могут стоять не только математические выражения, но и конкретные допустимые значения или константы (например, 2.70 или 3.5 и др.). В нашем случае всем переменным можно присвоить и значения целого типа, например, 2002 или 52. Это оказывается возможным, поскольку все целые числа входят в класс вещественных чисел.

1.3.10. Текстовая константа

Текстовые константы, например, “август 2002 года” и “этот текст является текстовой константой” ни одной из введенных переменных присвоить не удастся, ведь преобразовать такие тексты в числа *невозможно*, *Builder* об этом заявит на этапе компиляции программы. Текст, заключённый в верхние кавычки «», называется текстовой константой, он может содержать пробелы, набирается, как латиницей, так и кириллицей (или буквами иного национального алфавита).

1.3.11. Многострочный комментарий и преобразование текстовых переменных в переменные вещественного типа

Первый и второй шаги алгоритма нашей функции связаны с преобразованием текстовых значений чисел в вещественные значения и присваивание их переменным вещественного типа. Поэтому следующий фрагмент кода функции следует предварить, например, таким комментарием.

```
/*Преобразовываем текстовые значения числа в вещественные значения и присваиваем их переменным вещественного типа */
```

Если комментарий невозможно поместить на одной строке, то начало такого *многострочного комментария* обозначается знаками «/*» (без пробелов), а конец – знаками «*/». Конечно, этим способом можно ввести комментарий и на одной строке или даже её части. Описанный способ оформления комментариев очень широко применяется для временного исключения фрагментов программы при её разработке и отладке. В этом случае говорят, что фрагменты кода программы были *закомментированы*, или исключены из работы программы.

В нашей функции текстовые значения свойства *Text* визуальных объектов *Cena* и *Kolichestvo* требуется преобразовать к вещественному типу и запомнить (записать) соответственно в переменных вещественного типа *fCena* и *fKolichestvo*.

Программный доступ к свойствам визуальных объектов в *Builder* осуществляется операцией *стрелка* (\rightarrow). Для ввода этой операции (или знака) используются знаки минус «-» и больше «>», записанные слитно, без пробела. Подробно о сущности этой операции будет рассказано на десятом уроке. С использованием упомянутой операции доступ к значениям, хранящимся в свойствах *Text* объек-

тов *Cena* и *Kolichestvo*, осуществляется так: *Cena*→*Text*, *Kolichestvo*→*Text*.

Для преобразования текстового значения в значение типа *float* имеется функция *StrToFloat* (*String to float* – текстовый тип в тип *float*). Аргументом этой функции должен быть текст, в котором все символы являются цифровыми. Поэтому текст *Миша*, *Мама*, *Даша* не подходит, *Builder* сразу же известит об ошибке. Эта функция возвращает значение типа *float*, её использование в нижеследующих операторах присваивания полностью решает задачи первого и второго шагов алгоритма разрабатываемой функции:

```
fCena= StrToFloat(Cena->Text);
fKolichestvo= StrToFloat(Kolichestvo->Text);
```

Запрограммировать третий шаг алгоритма вы вполне сможете самостоятельно, у вас должен получиться, например, такой фрагмент кода функции:

```
//Вычисляем стоимость покупки
fStoimostj= fCena*fKolichestvo;
```

1.3.12. Преобразование переменных вещественного типа в переменные текстового типа

Для преобразования полученного результата из вещественного типа в текстовый, удобнее всего использовать функцию *FloatToStr* (переменная типа *float* в текстовую переменную). Её аргументом является переменная типа *float*, а возвращает она значение текстового типа. После преобразования переменной *fStoimostj* в текстовый тип её значение необходимо записать в свойстве *Caption* визуального объекта *Stoimostj*. В окончательном виде последний фрагмент кода функции может выглядеть так:

```
/*Преобразовываем вещественное значение к
текстовому значению и выводим его
в поле вывода Стоимость */
Stoimostj->Caption= FloatToStr(fStoimostj);
```

Приведём теперь полный текст разработанной функции *YmnogenieClick*.

```
void __fastcall TYmnogitel::YmnogenieClick(TObject *Sender)
{
    //Определяем переменные вещественного типа
    float fCena, fKolichestvo, fStoimostj;
    /*Преобразовываем текстовые значения числа в вещественные значения и
    присваиваем их переменным вещественного типа */
    fCena= StrToFloat(Cena->Text);
    fKolichestvo= StrToFloat(Kolichestvo->Text);
    //Вычисляем стоимость покупки
    fStoimostj= fCena*fKolichestvo;
    /*Преобразовываем вещественное значение к
    текстовому значению и выводим его
    в поле вывода Стоимость */
    Stoimostj->Caption= FloatToStr(fStoimostj);
} //YmnogenieClick
```


1.3.13. Форматирование текста программы

Каждый оператор присваивания, открывающая и закрывающая скобки функции *YmpogenieClick* находятся на *отдельной* строке, *все строки* кода и комментарии начинаются не с начала строки, а со столбца, смещённого на три знакоместа. Такой стандарт оформления текстов применяется в *структурном программировании*. Он позволяет оформлять программные коды в виде легко понятном не только на момент разработки, когда разработчик ещё хорошо помнит каждую деталь, но через месяцы и годы. Более того, любой читатель кода может быстро в нём разобраться (для изучения, модернизации или осознанного заимствования).

В журналах, книгах и газетах широко используют абзацные отступы, пропуски строк, выделение фрагментов текста повышенной жирностью, иным размером или видом шрифта, наклоном или подчёркиванием букв и многое другое. Большие произведения разбивают на отдельные тома, части, главы (или разделы), параграфы (или подразделы), пункты и подпункты. В конечном итоге это делается для более ясного и быстрого восприятия информации, содержащейся в тексте.

Аналогичные требования предъявляются и к текстам программ. Конечно, возможности форматирования здесь более ограничены, чем в полиграфии, однако они всё же имеются и ими неразумно пренебрегать. Во всех программах настоящего пособия используется форматирование. В программировании такое форматирование называется *структурированием программ*.

Структурирование также направлено на то, чтобы всю программу представить в виде отдельных законченных блоков или фрагментов. Каждый блок имеет название-комментарий и решает конкретные задачи. Любой из этих блоков можно использовать в других программах, где возникают такие же задачи. Работа блока реализуется, как правило, различными способами, поэтому его наполнение не является единственно возможным. Всё это позволяет складывать программы из таких блоков аналогично тому, как дом сложен из кирпичей. При этом из *одних и тех же* «кирпичей» можно сложить *разные* по назначению дома-программы; *одинаковые* по выполняемым задачам дома-программы могут быть построены из «кирпичей» *разного* вида. Такой метод ускоряет разработку программ, облегчает её изучение.

Писать хорошо структурированные коды следует приучаться с первых шагов программирования. Структурирование – это важный пункт хорошего стиля программирования.

В ходе разработки приложения для структурирования его кода часто бывает необходимо сдвинуть *серию* строк влево или вправо на несколько знакомест. Для ускорения такой сдвижки можно выполнить следующие действия: в *Редакторе Кода* выделить требуемый блок текста; нажать и не отпускать клавишу *Ctrl*; последовательно нажимать клавиши *k* и *i* для сдвига всего блока *вправо* на одно знакоместо или клавиши *k* и *u* – для сдвига блока *влево* на одно знакоместо.

Для тех, кто упорствует в приятии рекомендаций по структурированию, предлагаем разобраться в следующем тексте этой же функции (в нём отсутствуют комментарии, большинство пробелов, нет отдельных строк для каждого опе-

ратора присваивания).

```
{float fCena, fKolichestvo, fStoimostj; fCena=StrToFloat(Cena
->Text); fKolichestvo=StrToFloat(Kolichestvo->Text);
fStoimostj=fCena*fKolichestvo; Stoimostj->Caption=FloatToStr(fStoimostj);}
```

Можно не вводить переменные *fCena*, *fKolichestvo*, *fStoimostj*, в этом случае текст функции будет ещё короче:

```
{Stoimostj->Caption=FloatToStr(StrToFloat(Cena->Text) *
StrToFloat(Kolichestvo->Text));}
```

После уменьшения числа строк функции более чем в шесть раз её работа не очень понятна. Однако функция при этом осталась вполне пригодной для полноценного выполнения всех возложенных на неё задач. Такой метод приводит к большой компактности программ, однако работают они лишь на несколько процентов быстрее, а время разработки и отладки *увеличится в десятки и сотни раз. Никто* этот код (либо его фрагмент) *не применит повторно*. Серьёзную большую программу с использованием этого метода разработать весьма трудно. Поэтому экономия памяти и борьба за быстродействие программ должны быть разумными.

1.4. Сохранение и отладка


1.4.1. Сохранение проекта

Приступим к операции, время выполнения которой давно назрело: зафиксируем результаты наших трудов на жёстком диске, поскольку пока они находятся лишь в оперативной памяти, не способной хранить информацию после отключения питания компьютера или перезапуске *Builder*.

Сразу же после открытия нового приложения рекомендуем выполнить операции по его сохранению, в ходе разработки проекта (при создании очередного объекта или написании нескольких строк кода программы) осуществляйте периодическое *текущее* сохранение. Такой режим работы сводит к минимуму потери интеллектуального труда программиста.

Все файлы проекта настоятельно рекомендуем хранить в отдельном каталоге (папке) с содержательным названием, которое надлежит набирать *только латиницей*. Весьма желательно создавать текстовый файл с описанием назначения проекта.

Для сохранения всего проекта имеется три способа:

- ☐ Запустить команду сохранения при помощи горячих клавиш *Ctrl+Shift+S*.
- ☐ Использовать кнопку  на *Панели Инструментов* (см. рис. 1.5), она называется *Save All* (Сохранить всё).
- ☐ Применить подпункт горизонтального меню *File/ Save All* (см. 2 на рис. 1.5).

В любом из этих вариантов *Builder* предложит выбрать каталог для сохранения главного файла программы (с функцией *WinMain*). Появится окно, показанное на рис. 1.8.

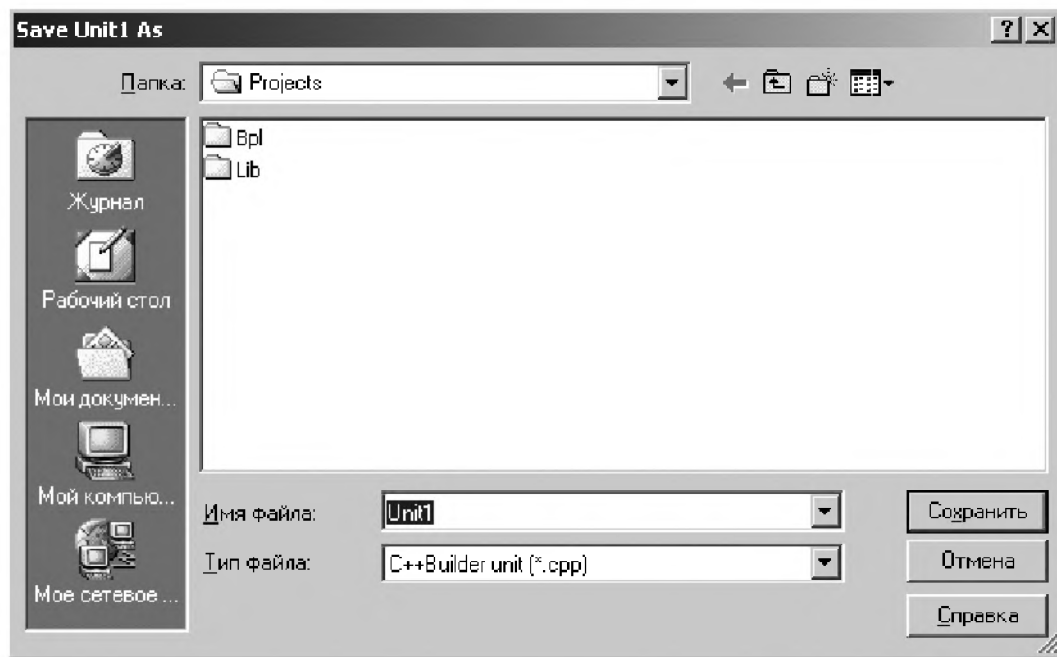


Рис. 1.8. Сохранение проекта

По умолчанию главный файл или модуль (*Unit*) программы получит имя *Unit1.cpp*. Согласно нашим рекомендациям создайте каталог *Yrok_1*. Затем имя *Unit1* замените на *Ymnogitel* и нажмите клавишу *Сохранить*.


Далее появится *новое* окно для сохранения файла с именем по умолчанию *Project1.bpr*. В нём находятся установки опций (параметры настройки) всех окон проекта и указания на то, какие файлы должны компилироваться и компоноваться в проект. Слово *Project* в названии файла дополним словом *Ymnogit*. Таким образом, полное новое имя файла *ProjecYmnogit.bpr*. Это необязательная рекомендация. Важно лишь, чтобы имена обоих файлов отличались друг от друга (и были информативными). В противном случае *Builder* выдаст грозное предупреждение и предложит придумать новое имя файла.

После операций сохранения в каталоге *Yrok_1*, помимо упомянутых файлов, будет находиться целый ряд других автоматически созданных (сгенерированных) файлов с различными расширениями.

1.4.2. Компиляция и сборка проекта

Переведём текст программы в машинный код. Эту работу выполняет специальная прикладная программа, называемая компилятором. Перед её запуском сохраните все изменения в вашем проекте командой *Save All*. Работа компилятора не сводится только к реализации упомянутой цели. Компилятор, прежде чем приступить к переводу текста программы в машинный код (трансляции, от *translate*) выяснит, нет ли в тексте программы синтаксических и некоторых других ошибок (о них расскажем позже).

Предположим, в тексте проекта нет никаких ошибок. Такая ситуация в программировании встречается *исключительно* редко. Пусть в вашей первой программе это произошло. Тогда, внесите в текст функции *YmnogenieClick* умышленную ошибку: прокомментируйте любым из двух способов точку с запятой после

первого оператора присваивания (или удалите этот знак). Далее откомпилируйте проект (запустите компилятор). Это можно сделать несколькими способами. Проще всего нажать горячие клавиши *Ctrl+F9*, или использовать горизонтальное (главное) меню: *Project/Make ProjectYmnogit* (Проект/Выполнить *ProjectYmnogit*). Процесс компиляции запускается также нажатием специальной кнопки . Для её помещения среди кнопок панели управления необходимо: в контекстном (или подручном) меню этой панели (для его вывода следует щёлкнуть *правой* кнопкой мыши по свободному от пиктограмм участку *Палитры Компонентов* или *Панели Управления*) выбрать пункт *Customize* (настройка); открыть вкладку *Commands* (команды), перейти в окно *Categories* (категории) и выбрать в нём опцию *Project*, а затем в её окне активизировать пункт *Make Project1* (компиляция проекта 1). После этого левой кнопкой мыши щёлкните в выбранном месте панели управления (или перетащите туда указанную пиктограмму).

Компилятор выведет вначале окно, информирующее о процессе фоновой компиляции. Однако если вы работаете с *Builder 5.0*, то это окно вы не успеете рассмотреть, поскольку, появившись, оно быстро исчезнет с экрана. После этого на экране выводится текст испорченной вами функции, а строка, над которой закомментирован разграничительный знак «;», выделяется красным цветом. В специальном окошке (оно расположено ниже окна *Редактора Кода*) указывается информация о сущности допущенной ошибки «*Statement missing ;*» (пропущен знак «;») и о месте её расположения в проекте (в нашем случае в тексте файла *Ymnogitelj.cpp* номер строки 24): *C++ Error] Ymnogitelj.cpp(24): E2379 Statement missing ;*. Компилятор нашёл ошибку, вот только место её расположения указал на строку ниже: он всегда указывает строку, *над* которой допущена ошибка, поскольку курсор устанавливается *после* ошибочного оператора. Окно, которое вы не успели рассмотреть в *Builder 5.0*, показано на рис. 1.9.

В его первой строке указано имя компилируемого проекта (и путь к нему). Вторая строка начинается словом *Done* (выполнено), в ней сообщается о том, что в тексте обнаружены ошибки (*There are errors*). В этой строке могут быть и другие

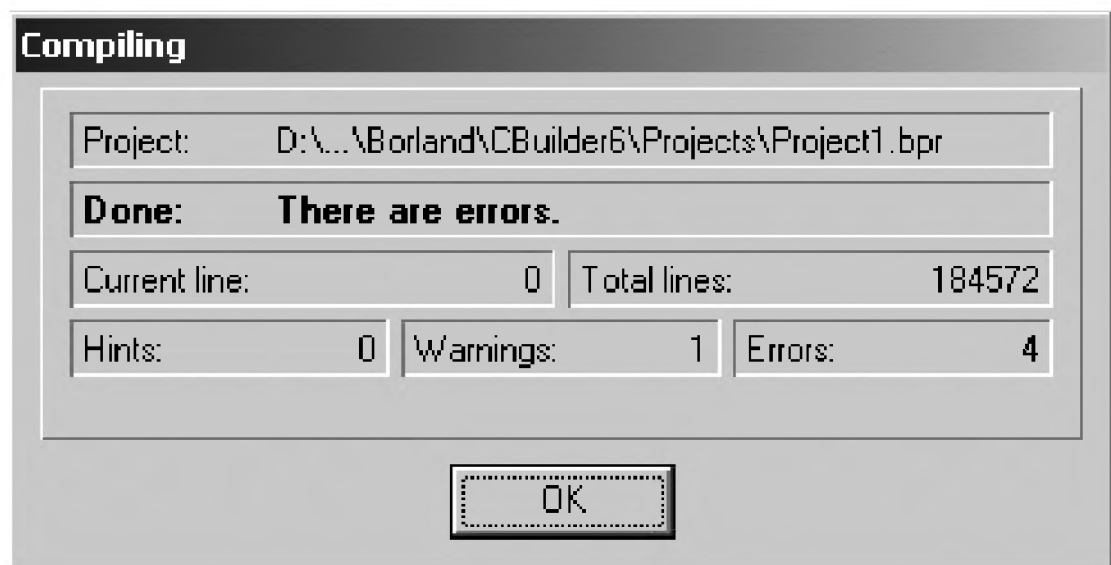


Рис. 1.9. Сообщение об итогах компиляции

сообщения (взамен указанного): *There are warnings* (имеются предупреждения), *Make* (исполнено). В полях вывода двух последних строк приведены: текущая компилируемая строка (*Current line*), суммарное число успешно откомпилированных строк (*Total lines*), число подсказок (*Hinds*), предупреждений (*Warnings*) и число обнаруженных ошибок (*Errors*). На второй строке окна может быть ещё строка с именем компилируемого модуля проекта. О предупреждениях и подсказках мы поговорим несколько позже, сейчас же отметим, что в *Builder 6.0* это окно после появления исчезнет только в результате нажатия кнопки *OK* или клавиши *Enter*. Для реализации этого режима в *Builder 5.0* (в *Builder 6.0* он включён по умолчанию) следует отключить опцию *Background Compilation* (фоновый режим компиляции). Для этого применяется команда горизонтального меню *Tools/Environment Option*, после чего на закладке *Preferences* необходимо снять птичку с опции *Background Compilation*.

Фоновый режим компиляции в ходе компиляции и компоновки проекта позволяет выполнять любые другие работы (подробнее о компоновке будет рассказано ниже). В этом режиме работа *Builder* осуществляется медленнее и окно с результатами компиляции исчезает автоматически спустя небольшой интервал времени. Необходимо отметить, что информация о допущенных ошибках дублируется в другом окне, которое никуда не исчезает и находится ниже окна *Редактора Кода*.

После исправления ошибки повторно запустите компилятор. Если вы не вводили новых ошибок, то в итоге получите сообщение о том, что программа успешно откомпилирована (*Make*). В результате в каталоге проекта создаётся объектный файл с расширением «.obj».

Полученный в результате компиляции объектный файл *ProjektYmnogit.obj* является промежуточным, поскольку он содержит только машинный код *вашей текстовой* части программы. Для создания на его основе исполняемого файла *ProjektYmnogit.exe*, к объектному коду необходимо подключить коды стандартных функций *Builder*, например, *StrToFloat()*. Такие функции хранятся в стандартных файлах-библиотеках с различными расширениями. В упомянутых библиотеках хранятся также математические функции, описание типовых объектов интерфейса, таких как поля, кнопки и др. Вставка в объектный код кодов стандартных функций называется *сборкой программы* или её *линковкой (linking)*. Эту работу выполняет специальная программа, она называется *линковщик*. Линковщик вставляет необходимые коды в соответствующие места объектного файла. За это его иногда называют редактором связей. В ходе компиляции осуществляется ещё так называемая препроцессорная обработка: выполняются директивы (указания) компилятору, подключаются имена заголовочных файлов. Более подробно об этом будет рассказано на последующих уроках. Исполняемый файл очень компактный по объёму, при специальном режиме компилятора (см. следующий урок) его можно запускать на выполнение на компьютере, на котором не установлен *Builder*. Ход создания исполняемого файла схематично показан на рис. 1.10.

После завершения компиляции надпись *Compiling* в окне (см. рис. 1.9) автоматически сменится на *Linking*. Это сообщение информирует вас о том, что к ра-

боте приступил линковщик. Однако ввиду того, что программа имеет очень малый объём, компилятор и сборщик выполняют свои задачи очень быстро, и поэтому всех этапов создания *exe*-файла вы в этот раз не заметите.

Компиляцию можно осуществить также командой *Project/ Build ProjectYmnogit*. Такая команда компилирует *все модули проекта*, независимо от того, когда они в последний раз изменялись. А вот рассмотренная нами выше команда *Project/ Make ProjectYmnogit* компилирует и собирает только те модули, которые в проекте были изменены с момента предыдущей компоновки проекта, поэтому, естественно, выполняется быстрее. Но для нашей первой очень маленькой учебной задачи, времени работы этих команд будут практически одинаковыми. Описав их назначения, мы вооружаем вас эффективными механизмами разработки ваших *будущих* проектов. Только в качестве подготовки вас к предстоящим подвигам уже сейчас сообщаем о том, что помимо рассмотренных команд компиляции существуют ещё две: *Project/Make All Project* и *Project/Build All Project*. Они подобны рассмотренным выше командам *Make* и *Build*, но относятся к работе с *группой* проектов, их действия распространяются на всю серию разрабатываемых связанных между собой проектов. Как видите, число неизвестных команд и пиктограмм *Builder* постепенно уменьшается.

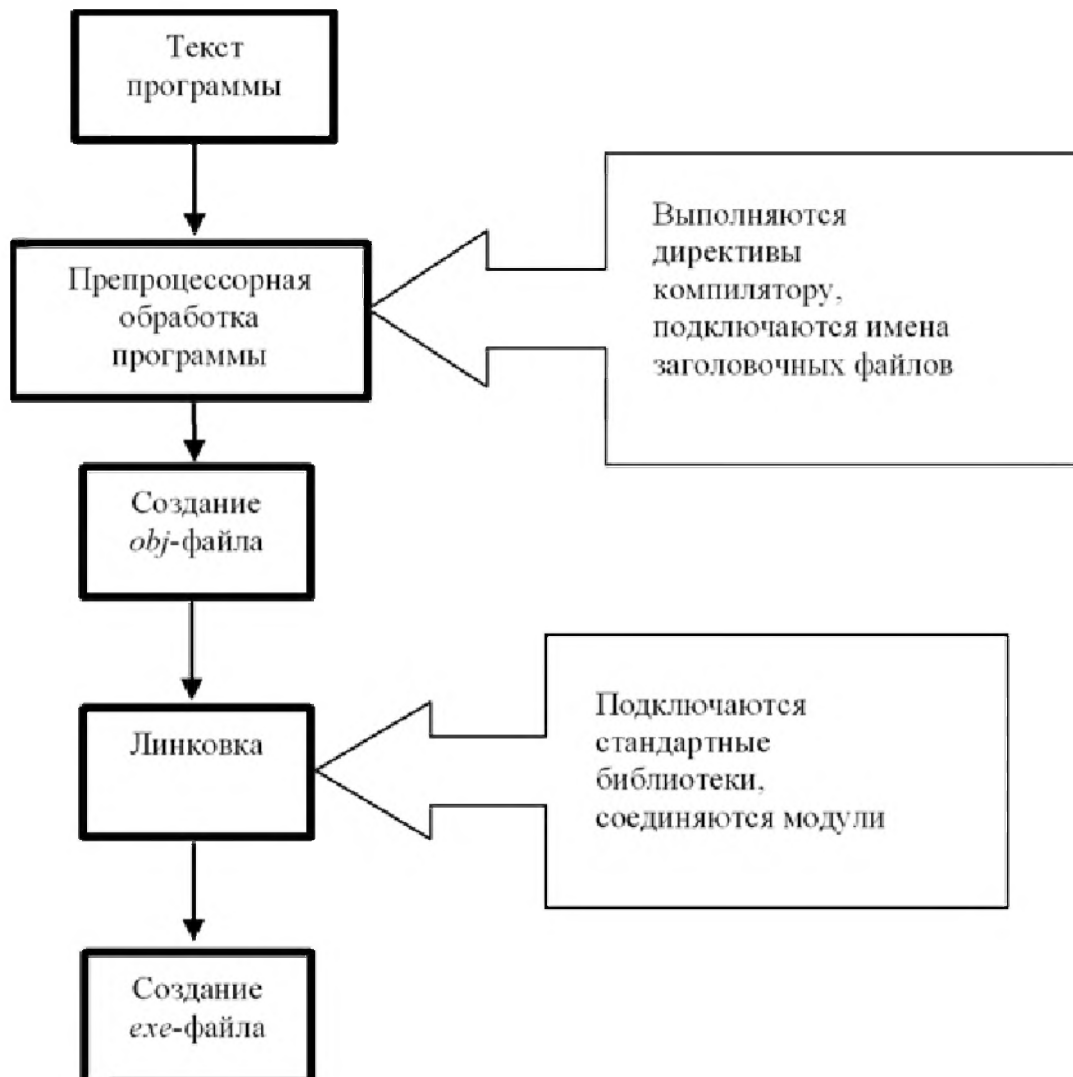


Рис. 1.10. Этапы компиляции программы

1.4.3. Предупреждения и подсказки

В результате компиляции не должны выводиться никакие предупреждения и подсказки. Поскольку, в противном случае нельзя гарантировать, что программа работает правильно. *Предупреждения* – это сообщения о потенциально опасных местах программы, которые вместе с тем не являются синтаксическими ошибками. Приведём поясняющий пример. Ниже показана часть кода функции *YmnogenieClick*:

```
float fCena, fKolichestvo, fStoimostj;  
fCena= StrToFloat(Cena->Text);  
fStoimostj= fCena* fKolichestvo;
```

При компиляции этого фрагмента будет выведено предупреждение: «[C++ Warning] *Ymnogitelj.cpp*(26): W8013 Possible use of 'fKolichestvo' before definition» (На 26 строке файла *Ymnogitelj.cpp* переменная *fKolichestvo* используется перед её инициализацией). Здесь к моменту перемножения переменных *fCena* и *fKolichestvo* значение переменной *fKolichestvo* ещё *не определено*, в ячейке памяти с упомянутым именем может храниться *произвольная* информация, поэтому и результат перемножения, значение в переменной *fStoimostj*, будет *непредсказуемым*. Такая оплошность может привести к длительному поиску ошибки. Наиболее опасным является правдоподобный результат, он чреват *ложными открытиями*. Следует *всегда* помещать *нужные* значения в переменные перед их использованием (например, в математических выражениях).

Подсказка направлена на оптимизацию работы программы, на устранение ненужных операторов. Приведём неоптимальный фрагмент кода нашей функции:

```
fCena= 2.70;  
fCena= StrToFloat(Cena->Text);
```

В этом случае компилятор выведет подсказку: «[C++ Warning] *Ymnogitelj.cpp*(23): W8004 'fCena' is assigned a value that is never used» (На 23 строке файла *Ymnogitelj.cpp* переменной *fCena* присваивается значение, которое нигде далее не используется).


Первый оператор присваивания является совершенно ненужным, поскольку его результаты *нигде* не используются, сразу же после его применения в ячейку памяти с именем *fCena* будет записана *новая* информация. Поэтому упомянутый оператор без какого-то ущерба следует исключить, что сократит время выполнения и объём памяти, занятый под размещение программы.

Предупреждения и подсказки по умолчанию в *Builder* не выводятся. Как осуществить вывод этой информации рассказывается на втором уроке.

1.4.4. Запуск проекта

Мы получили исполняемый файл *ProjektYmnogit.exe*. Его можно запустить на выполнение в любом файловом менеджере (*FAR*, *Windows Commander*, проводнике и др.), а также из командной строки или раздела *Выполнить* кнопки *Пуск*. Если

вы пожелаете запустить его с командной строки, то в файловом менеджере директорию с учебным проектом необходимо сделать текущей. Однако без специальной предварительной настройки параметров *Builder* (предпринятой перед получением *ProjektYmnogit.exe*) после переноса файла на компьютер, на котором не установлен *Builder*, этот файл запускаться не будет. Почему так происходит и что требуется сделать для его полной универсальности, рассказывается на следующем уроке. Сейчас же поработайте с готовой программой, введите несколько пар исходных значений, удостоверьтесь в том, что она работает правильно.

В заключение настоящего пункта необходимо отметить, что запустить *exe*-файл на выполнение можно и из среды разработки программы. Именно так и нужно поступать *в ходе работы* над проектом, поскольку при этом вы не выходите из среды программирования, что экономит время. Для запуска исполняемого файла из среды программирования *IDI* следует нажать клавишу *F9*, либо ввести команду горизонтального меню *Run/ Run*. Можно также запуск программы осуществить кнопкой на панели управления с зелёным треугольником. Эта кнопка выглядит вот так . Если вы запустите на выполнение ещё не откомпилированный проект, то *Builder* вначале последовательно (с вашим участием) выловит все ошибки программы, затем её откомпилирует, соберёт и только после этого запустит исполняемый файл. Различие команд компиляции и запуска программы (в среде программирования) лишь в том, что *после компиляции* исполняемый файл *не запускается* на выполнение. Мы не рекомендуем запускать проект клавишей *F9* до её успешной компиляции командой *Ctrl+F9*.

Закончена разработка первого проекта. На его основе вы познакомились с большим объёмом фундаментальных знаний. Перед следующим шагом в программировании предлагаем ответить на вопросы для самоконтроля, они помогут упорядочить полученные знания, проверить, насколько осознан ваш первый шаг.

Вопросы для самоконтроля

- ☐ С какой целью используются комментарии в программе, при помощи каких знаков они вводятся?
- ☐ Что называют правильным идентификатором?
- ☐ В чём заключается сущность метода венгерской нотации?
- ☐ Опишите процесс компиляции и сборки программы.
- ☐ Как объявляются и инициализируются переменные?
- ☐ Назовите все элементы оператора присваивания.
- ☐ Укажите общие и отличительные черты команд, вводимых при помощи функциональных клавиш *F11* и *F12*.
- ☐ В каком случае команда *F9* выполняется также как и команда *Ctrl+F9*?
- ☐ Как записать на жёсткий диск все файлы проекта?

1.5. Задача для программирования

Разработайте калькулятор, способный выполнять лишь четыре арифметические операции над двумя числами, вводимыми в поля ввода *x* и *y*. Результат вычисле-

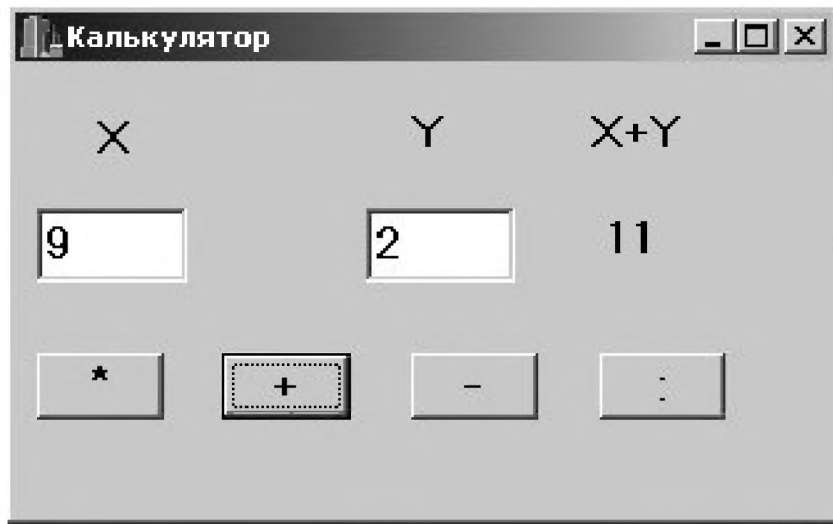


Рис. 1.11. Интерфейс программы Калькулятор

ния и подсказка выбранного действия должны показываться после нажатия требуемой командной кнопки. Интерфейс проекта может выглядеть, например, так, как показано на рис. 1.11. Открытие окна для разработки *нового* приложения (*aplication*) в *Builder 5.0* осуществляется командой *File/New Aplication*, а в *Builder 6.0* – командой: *File/New/Application*. Действия «сложение», «вычитание» и «деление» в *C++* выполняются соответственно при помощи знаков «+», «-» и «/».

1.6. Вариант программного решения

Здесь приведен только файл реализации с функциями по обработке нажатий клавиш.

```
//Эта часть файла генерируется автоматически
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TCalculator *Calculator;
__fastcall TCalculator::TCalculator(TComponent* Owner)
    : TForm(Owner)
{
}
//Все тела нижеследующих функций
// разрабатываются программистом

void __fastcall TCalculator::YmnogenieClick(TObject *Sender)
{
    float fx, fy, fz;
    fx= StrToFloat(x->Text);
    fy= StrToFloat(y->Text);
    fz= fx*fy;
    z->Caption= FloatToStr(fz);
    Operac->Caption= "X*Y";
} //YmnogenieClick
```

```
void __fastcall TCalculator::SlogenieClick(TObject *Sender)
{
    float fx, fy, fz;
    fx= StrToFloat(x->Text);
    fy= StrToFloat(y->Text);
    fz= fx+fy;
    z->Caption= FloatToStr(fz);
    Operac->Caption= "X+Y";
} //SlogenieClick

void __fastcall TCalculator::VuchetanieClick(TObject *Sender)
{
    float fx, fy, fz;
    fx= StrToFloat(x->Text);
    fy= StrToFloat(y->Text);
    fz= fx-fy;
    z->Caption= FloatToStr(fz);
    Operac->Caption= "X-Y";
} //VuchetanieClick



void __fastcall TCalculator::DelenieClick(TObject *Sender)
{
    float fx, fy, fz;
    fx= StrToFloat(x->Text);
    fy= StrToFloat(y->Text);
    fz= fx/fy;
    z->Caption= FloatToStr(fz);
    Operac->Caption= "X:Y";
} //DelenieClick
```

Урок 2. Модернизация программы

В проекте *Умножитель* результат вычислений представим в более удобном для восприятия виде, обеспечим проверку вводимых данных, сделаем полноценным исполняемый файл проекта (обеспечим его запуск на компьютерах, на которых не установлен *Builder*) и познакомимся с инструментарием по улучшению интерфейса.

2.1. Открытие проекта

Для внесения усовершенствований необходимо найти и запустить основной файл проекта (с расширением «.bpr»). Вывод окна поиска осуществляется одним из способов:

- ☐ Командой *File/Open Project* (*Файл/Открыть проект*).
- ☐ Комбинацией клавиш *Ctrl+F11*.
- ☐ При помощи пиктограммы .
- ☐ Команда *File/Reopen* (*Файл/Открыть повторно*) выводит список проектов, с которыми работали последнее время.
- ☐ Эта же команда вводится нажатием кнопки раскрывающегося списка пиктограммы .
- ☐ Командой *Пуск/Документы* выводится список документов и проектов, с которыми недавно работали.

2.2. Округление результата

Запустите программу *Умножитель*, в полях ввода введите цену товара 12,53 и его количество 2,77. Нажав кнопку *Умножение*, в поле *Стоимость* увидите сумму 34,708099365. Часто столь большое количество цифр после запятой является излишним, оно лишь ухудшает восприятие числа. Кроме того, в типе *float* воющая и последующие цифры неверны. В нашем случае вполне достаточно округлить результат лишь до двух цифр после запятой.

Функция *Builder FloatToStrF()* округляет вещественное число до заданного количества цифр после запятой и преобразует его к текстовому формату. Её аргументом может быть число произвольного вещественного типа (не только типа *float*). Все возможности упомянутой функции описаны в приложении к настоящему разделу, с её использованием последний фрагмент функции *YmnogenieClick* выглядит так:

```
/*Преобразовываем вещественное значение переменной в  
текстовое и выводим его в поле вывода Стоимость */
```

```
Stoimostj->Caption= FloatToStrF(fStoimostj, ffFixed, 10, 2);
```

ffFixed – это константа, определяющая формат с фиксированной точкой (запятой) вида «*ddd.ddd...*». По крайней мере, одна цифра *всегда* предшествует десятичной точке. Количество цифр после десятичной точки здесь задаётся числом «2», в общем случае оно может находиться в пределах от 0 до 18 (например, 12345.689). Если количество цифр слева от десятичной точки окажется больше параметра «10», то автоматически будет применён *научный* или *экспоненциальный* формат (1234.5678E+12). Вместо «10» могут стоять числа от 1 до 18.

Для проверки работы модернизированного фрагмента функции *YmnogenieClick* сделайте новую директорию с именем *Yrok_2*, скопируйте в неё все файлы проекта *Ymnogitel*, затем внесите упомянутые исправления в функцию *YmnogenieClick*, сохраните, откомпилируйте и запустите проект *Ymnogitel*. Убедитесь в том, что приложение работает правильно: значения результата округляются до двух цифр после запятой.

Приложение 2.1. Возможности функции *FloatToStrF()*

С этим материалом можно ознакомиться при повторном чтении пособия.

Функция *FloatToStrF(Chislo, Format, Tochnostj, Cifru)* выполняет преобразование вещественных чисел:

- ❑ *Chislo* – любое вещественное число;
- ❑ *Format* – одна из констант форматирования (*ffExponent*, *ffFixed*, *ffCurrency*, *ffGeneral*);
- ❑ *Tochnostj* – в формате *ffExponent* это *общее* число цифр (включая одну перед десятичной точкой). В формате *ffFixed* аргумент *Tochnostj* определяет количество целых цифр числа, при которых число выводится в формате с фиксированной точкой, если количество цифр в выводимом числе окажется больше *Tochnostj*, то тогда число выводится в формате *ffExponent*;
- ❑ *Cifru* – в форматах *ffFixed* и *ffCurrency* это количество цифр после запятой, которые вы желаете оставить (при этом производится округление последней оставляемой цифры), изменяется в пределах от 0 до 18; в формате *ffExponent* это количество цифр, определяющее число разрядов степени числа 10, изменяется от 0 до 4.

Параметр *Format* задаётся одной из пяти системных переменных типа *enum* (от *enumeration* – перечисление, этот тип переменных рассматривается в разделе 13.1):

- ❑ *ffExponent* – научный или экспоненциальный формат, в котором число преобразуется к виду «*-d.ddd...E+dddd*». Общее число цифр (до символа «E»), включая одну перед десятичной точкой, задаётся параметром *Tochnostj*. После символа «E» *всегда* следует знак «+» или «-», после этих знаков может стоять до четырёх цифр. Параметр *Cifru* определяет число разрядов степени числа 10 и изменяется от 0 до 4. Пример: «12345.6789E+1234=12345.6789Ч10¹²³⁴».
- ❑ *ffFixed* – формат с фиксированной точкой вида «*ddd.ddd...*». По крайней

мере, одна цифра *всегда* предшествует десятичной точке. Число цифр после десятичной точки задаётся параметром *Cifru*, который может изменяться в пределах от 0 до 18. Пример: «12345.68». Если количество цифр слева от десятичной точки больше параметра *Tochnostj*, то используется научный формат.

- ❑ *ffGeneral* – основной числовой формат. Число преобразуется к формату с фиксированной точкой (*ffFixed*) или научному (*ffExponent*) в зависимости от того, какой из них имеет меньшее число символов (короче). При этом начальные нули числа удаляются, десятичная точка ставится только при необходимости. Если число разрядов числа слева от точки не больше параметра *Tochnostj* или значение числа не меньше 0.00001, то используется фиксированный формат. В противном случае применяется научный формат, в котором параметр *Cifru* определяет число разрядов степени числа 10 и изменяется в диапазоне от 0 до 4.
- ❑ *ffNumber* – числовой американский формат, в котором число преобразуется в строку вида «-d,ddd,ddd.ddd...». Данный формат совпадает с фиксированным форматом (*ffFixed*) за исключением наличия в нём разделителей классов (тысяч), широко применяемых в США.
- ❑ *ffCurrency* – монетарный формат для преобразования числа в строку, отображающую денежную сумму. Число цифр после десятичной точки задаётся параметром *Cifru*, который в данном формате может лежать в пределах от 0 до 18. Этот формат контролируется глобальными переменными, задаваемыми для монетарного формата разделом *Currency* (Виды валют) элемента *International* (Международный).

2.3. Борьба с ошибками пользователя

2.3.1. Исключительные ситуации

Как отмечалось на прошлом уроке, для набора вещественных значений в *теле программы* используется *разделительная точка*, а в *полях ввода* интерфейса применяется *разделительная запятая*. Но отечественные пользователи ваших программ могут ошибочно заполнить поля ввода данными с применением разделительной точки, либо ввести не числа, а слова, например, «одиннадцать с половиной». В этом случае функция *StrToFloat()* не может преобразовать такой нечисловой текст в число, и поэтому дальнейшая работа программы невозможна. Также вычисления не выполняются, если введенное число выходит за границы использованного типа данных, при делении на нуль.

Во всех случаях, когда какая-либо стандартная функция или операция не может выполнить свои задачи (из-за некорректности аргумента, аргументов или данных) происходит остановка работы программы, возникает *прерывание программы* (или просто *прерывание*) или *исключительная ситуация*.

У *Builder* для каждой исключительной ситуации имеется свой класс или тип.

Название класса, предназначенного для обработки прерываний, всегда начинается с прописной буквы «E» (от *Exception* – исключение). Всего таких классов – несколько десятков.

При помощи справочной системы (*Help*) можно выяснить какие исключительные ситуации могут возникать при работе с той или иной стандартной функцией *Builder* (исключения имеются не у всех функций). Приведём *наиболее* удобный способ вызова справки.

Сделайте активным окно *Редактора Кода*, установите курсор на первой букве её имени (или выделите подсветкой имя требуемой функции) и нажмите функциональную клавишу *F1*. В результате появится статья справочной системы, посвящённая запрашиваемой функции.

Например, в последнем предложении справки для функции *StrToFloat* сообщается, что при некорректности её параметра возникает *единственная* исключительная ситуация *EConvertError*.

Если в функции реакция на исключительные ситуации *явно* не предусмотрена, то при их появлении всегда вызывается *стандартный* обработчик прерываний. Такая ситуация возникнет, в частности, когда в программе произведена попытка деления на нуль. Некоторые функции при некорректных значениях своих аргументов возвращают неопределённое значение, информирующее *программиста* об ошибке. Например, функция *FloatToStrF()* при невозможности преобразования числа в текст возвращает сообщение *NAN* (*not a number* – нет числового значения).

Таким образом, *Builder* без участия программиста может обрабатывать исключительные ситуации. Покажем эту возможность на примере. В среде разработки программ запустите приложение *Умножитель* (нажмите клавишу *F9*). Далее в поле ввода *Цена* наберите нечисловое значение, например, «23.3», а в поле *Количество* – «4». После нажатия кнопки *Умножение разработчику* программы в



Рис. 2.1. Стандартное окно, информирующее о причине прерывания

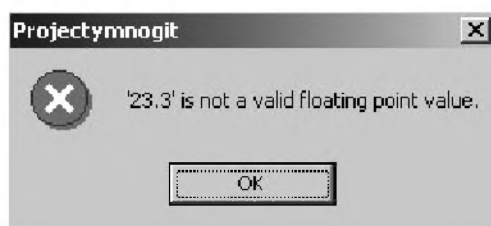


Рис. 2.2. Информация для пользователя программы о допущенной им ошибке

специальном окне *подробно* сообщается о сущности исключения. Это окно показано на рис. 2.1.

Для продолжения работы программы нажмите вначале кнопку *ОК* (на этом окне), а затем клавишу *F9* (или введите команду *Run/Run* горизонтального меню). В результате появится новое окно, *кратко* информирующее *пользователя* программы о сути ошибки. Это окно показано на рис. 2.2.

Корректно завершать работу программы следует горячими клавишами *Ctrl+F2* (или командой горизонтального меню *Run/Program Reset* (Завершить программу)). В этом случае *Builder* вначале освободит использованные ячейки памяти и другие системные ресурсы, а затем закроет окно запущенной программы и покажет её текст в *Редакторе Кода*.

Если же вы запустите приложение *Умножитель* не в *Builder*, а при помощи проводника, *FAR*, пункта *Выполнить* меню кнопки *Пуск* или командной строки в *FAR*, то после ввода нечислового значения и нажатия командной кнопки появится *только второе* окно, предназначенное для пользователя (а не программиста). Повторяем, первое окно с подробной информацией об ошибке появляется *только* в *IDE*. Оно предназначено для разработчика программы с тем, чтобы он предпринял необходимые меры по обработке этой исключительной ситуации.

Можно, в принципе, не изменять текст, который выводится в окне на рис. 2.2, поскольку он очень понятен, его перевод: «23.3 невозможно преобразовать в вещественное число». Однако, для русскоязычных пользователей (не владеющих английским), обеспечим вывод таких сообщений *на великом и могучем*.

С этой целью фрагмент функции *YmnogenieClick*, предназначенный для *корректной* обработки любого введенного значения (как правильного, так и ошибочного) может выглядеть так:

```
try
{
    fCena= StrToFloat(Cena->Text);
} //try
catch (const EConvertError &)
{
    ShowMessage("В поле \"Цена\" введено нечисловое значение");
} //catch
```

В приведенном фрагменте после служебного слова *try* (пытаться, пробовать) имеется *логический блок* с прежним оператором присваивания «*fCena=StrToFloat(Cena->Text);*». Далее на отдельной строке приведено ключевое слово *catch* (схватить, поймать, обнаружить) с параметром в круглых скобках: «*const EConvertError &*». После этой строки поставлен *логический блок* с функцией *ShowMessage()*. Заметим, что служебное слово *const* в круглых скобках можно не указывать, после знака амперсанта & (с пробелами или без них) можно поставить прописную или строчную букву *E*.

При наличии такого фрагмента функция *YmnogenieClick* *попытается* вначале преобразовать текст в переменную вещественного типа с использованием оператора, находящегося в первом логическом блоке. При помощи служебного слова *try* над *всем* этим блоком установлен контроль. В случае, когда преобразование

выполнено успешно, строка *catch* (*const EConvertError &*) и второй логический блок игнорируются (пропускаются). Управление программой передаётся оператору, расположенному сразу же за вторым логическим блоком.

Если же упомянутое преобразование невозможно, то осуществляется работа операторов второго логического блока (в нашем примере там находится один оператор). Он предназначен для *обработки* ситуации *EConvertError*. Как отмечалось выше, обработка данного прерывания заключаться лишь в выводе окна с соответствующей информацией на русском языке. Для вывода такого окна используется функция *ShowMessage()* (показ сообщений). Её единственным параметром является текстовое значение. В нашем случае это текстовая константа *"В поле \"Цена\" введено нечисловое значение"*. В *Builder* существуют и другие функции для вывода сообщений.

Но почему двойные кавычки, необходимые в *C++* для окаймления всех текстовых значений, находятся не только по краям сообщения, но и внутри него? Это сделано для того, чтобы в сообщении слово *Цена* выводилось в двойных кавычках. Если перед кавычками стоит символ «\» (обратная косая черта или левый слеш), то такие кавычки не являются началом или концом текстовой константы.

Теперь, после запуска программы (в *IDE*), при нечисловом значении цены на экране появится вначале окно, показанное на рис. 2.1, а затем (после нажатия кнопки *OK* и клавиши *F9*) – окно, приведенное в левой части рис. 2.3.

При запуске программы не с *IDE*, на экране появится только второе окно. В его заголовке по умолчанию выводится имя исполняемого файла проекта. Для изменения заголовка введите команду *Ctrl+Shift+F11* (или *Project/Option*), далее на вкладке *Application* (см. рис. 2.4) в окне *Title* (заголовок) наберите новый заголовок, например, *Умножитель*. Окно с таким заголовком показано в правой части рис. 2.3.

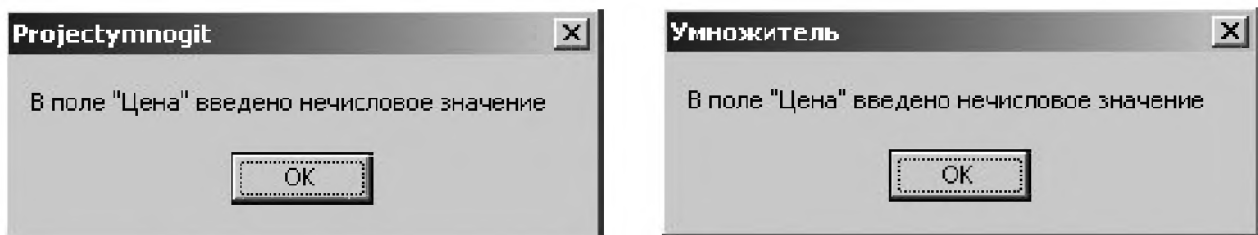


Рис. 2.3. Результат корректной обработки исключительной ситуации

В приложениях часто возникают исключительные ситуации, которые обрабатываются при помощи таких констант:

- ❑ *EOverflow* – слишком большое или слишком маленькое число, выход за границы применяемого типа;
- ❑ *EZeroDivide* – деление на нуль вещественного числа;
- ❑ *EDivByZero* – деление на нуль целого числа.

После *try*-блока может следовать *несколько catch*-блоков, каждый из которых предназначен для обработки *конкретного* прерывания, указанного соответствующим

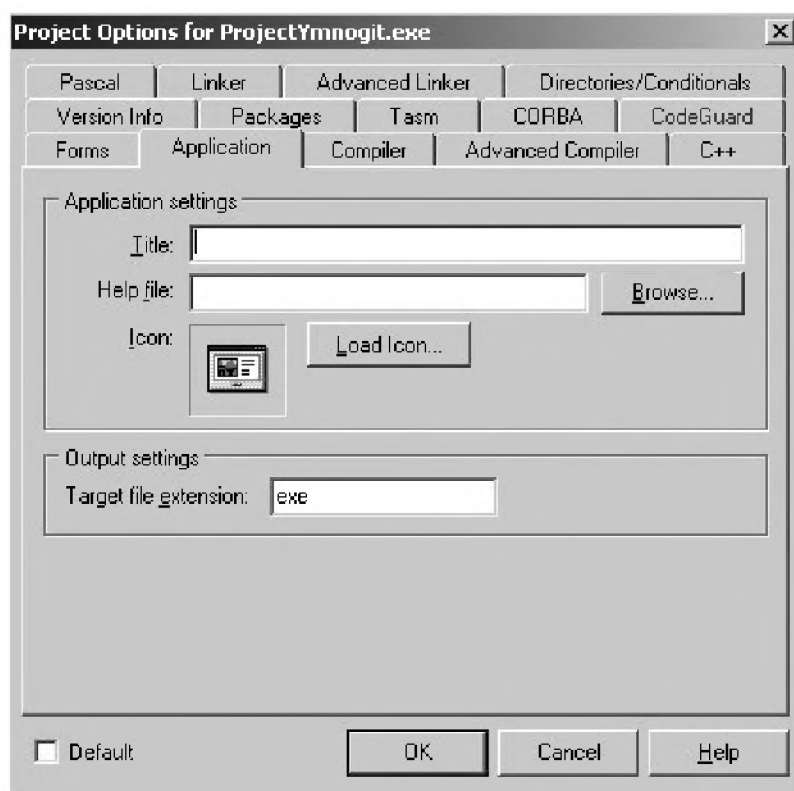


Рис. 2.4. Вкладка Application

щей константой. Число *catch*-блоков ограничивается лишь *суммарным числом* исключительных ситуаций для *всех* функций блока, над которым установлен *try*-контроль (в нашем примере в блоке находится только одна функция).

Если вы сомневаетесь в том, что создали абсолютно все *catch*-блоки для обработки исключительных ситуаций *try*-блока, то в круглых скобках последнего *catch*-блока поставьте лишь три точки:

```
catch (...)
{
    ShowMessage("Произошла ошибка");
} // catch
```

Этот блок перехватывает *все* ошибки блока *try*, что приводит к блокировке последующих за ним *catch*-блоков (если они имеются). Поэтому такой блок всегда должен *стоять последним*. При игнорировании упомянутого требования, в результате компиляции получите сообщение: «[C++ Error] Ymnogitelj.cpp(31): E2174 The '...' handler must be last» (обработчик «...» должен стоять последним).

Если в ходе работы фрагмента программы с несколькими *catch*-блоками (относящимися к единственному *try*-блоку) один из *catch*-блоков окажется выполненным, то вне зависимости от того, где он стоит (первым или последним), управление программой передаётся первому оператору, стоящему за *последним catch*-блоком.

Проверка правильности ввода данных в поле ввода *Количество* осуществляется аналогичными *try-catch*-блоками. Число *try-catch* блоков в программах не ограничивается. Далее приведём полный фрагмент функции, отвечающий за ввод данных.

```
/* Преобразовываем текстовые значения в вещественные и присваиваем их
   переменным вещественного типа/
try
{
    fCena= StrToFloat(Cena->Text);
} //try
catch (const EConvertError &)
{
    ShowMessage("В поле \"Цена\" введено нечисловое значение");
} //catch
catch (const EOverflow &)
{
    ShowMessage("Введено слишком большое или слишком маленькое\
                значение ");
} //catch

try
{
    fKolichestvo= StrToFloat(Kolichestvo->Text);
} //try
catch (const EConvertError &)
{
    ShowMessage("В поле \"Количество\" введено нечисловое значение");
} //catch
catch (const EOverflow &)
{
    ShowMessage("Введено слишком большое или слишком маленькое\
                значение ");
} //catch
```

Если в оба поля ввода вписать ошибочные значения, то программа последовательно выведет сообщения об этих ошибках.

Программист первостепенное внимание должен уделять экономному использованию памяти компьютера, быстродействию и надёжности программ. Последнее требование является самым важным. Поэтому не беспокойтесь из-за резкого увеличения числа строк функции, поскольку при этом надёжность работы программы повышается.

2.3.2. Молчаливые исключения

После обработки исключения происходит его *разрушение* (устранение), в результате управление программой передаётся оператору, стоящему за последним *catch*-блоком. Для получения какого-то результата в *catch*-блоках *можно* предусмотреть замену ошибочных данных допустимыми значениями (помимо вывода предупреждения об ошибке). Однако в учебной программе в *catch*-блоках ошибки в данных *не исправлялись* и поэтому *никакие* последующие операции с ними *невозможны*. Почему же не происходили прерывания на этих операциях?

У *Builder* имеется исключение *EAbort* (*abort* – аварийный, преждевременный выход), при возникновении которого остановка программы происходит без вывода специального диалогового окна, происходит *молчаливое* прерывание. Если обычные прерывания устраняются нажатием кнопки *ОК* появившегося окна, то

молчаливые прерывания разрушаются *автоматически* сразу же после их появления. Таким образом, при операциях, *невыполнимых* из-за неопределённости их параметров, возникают прерывания типа *EAbort*. В результате управление программой последовательно перемещается с первой такой операции на последнюю через все промежуточные операции (или операторы) программы. Для пользователя же приложения этот процесс выглядит как *немедленный* выход из программы после *ручного* разрушения последнего исключения.

2.4. Улучшение интерфейса

2.4.1. Выбор пиктограммы для приложения

Каждому исполняемому файлу приложения *Builder* назначает одну и ту же пиктограмму, установленную по умолчанию. Для её замены необходимо:

- ☐ Ввести команду *Ctrl+Shift+F11* (или *Project/Option*).
- ☐ Выбрать вкладку *Application*.
- ☐ На этой вкладке (см. рис. 2.4) нажать кнопку *Load Icon...* (Загрузить пиктограмму) или *Browse...* (Выбор файла).
- ☐ В появившемся стандартном окне *Windows*, предназначенном для выбора файла, найти подходящий файл-картинку (с расширением *ico*) по адресу: *C:\Program Files\Borland\CBuilder5(или CBuilder6)\ObjRepos(или Examples\Icon)*. Позже картинки-пиктограммы научитесь создавать самостоятельно.
- ☐ Щёлкнуть по выбранному значку, затем по кнопке *Открыть*.
- ☐ Нажать кнопку *OK* и перекомпилировать проект.

2.4.2. Местоположение интерфейса

После запуска приложения его интерфейс располагаться в том месте экрана, где находилась форма при её проектировании. Для помещения интерфейса в *центре* экрана необходимо в *Инспекторе Объектов* при помощи раскрывающегося списка выбрать имя формы *Ymnogitel* (укажите второй способ такого выбора) и её свойству *Position* (положение) из списка предлагаемых значений назначить значение *poScreenCenter* (по центру экрана). У этого свойства имеется целый ряд других значений, полная информация об их назначении указана в справочной статье, вызываемой клавишей *F1*.

2.4.3. Плавное перетаскивание и протягивание

При перетаскивании объектов формы, они не всегда располагаться в указанном месте. Как правило, они прыгают по горизонтали и вертикали на постоянное небольшое расстояние, называемое шагом сетки (расстояние между ближайшими точками формы – узлами сетки). Точки на форме (на интерфейсе их нет) нужны лишь для удобства размещения на ней объектов, поскольку они автоматически привязываются к *ближайшим* узлам сетки (по умолчанию).

Если привязка объектов к узлам сетки не нужна, то перед их буксировкой нажмите клавишу *Alt* и не отпускаете её до завершения перетаскивания. Аналогичным способом осуществляется плавное изменение габаритов объектов.

Для достижения этих же целей можно отключить режим *Snap to grid* (Привязка к сетке) в окне, вызываемом командой *Builder 5.0: Tools/Environment Option/Preferences* (Инструменты/ Параметры среды/Установки). Эта же команда в *Builder 6.0* выглядит несколько по-иному: *Tools/Environment Option/Preferences/Designer (Дизайнер)*. Окно для отключения указанного режима показано на рис. 2.5.

В этом же окне шаг сетки по горизонтали и вертикали изменяется от 2 до 128 пикселей. Для этого используются поля *Grid size X* (Шаг по горизонтали) и *Grid size Y* (Шаг по вертикали). При минимальном шаге прыжки объектов малозаметны. Режимом *Display grid* (Показать сетку) позволяет включить/отключить точки сетки на форме.

2.4.4. Горизонтальное и вертикальное выравнивание

Выравнивание объектов формы вдоль горизонтальной линии осуществляется следующими операциями:

- ❑ Выведите контекстное меню выравниваемых объектов. Для этого нажмите левую кнопку мыши в любом углу прямоугольной области, внутри которой будут заключены выравниваемые объекты и, не отпуская её, протяните указатель мыши к противоположному углу (по диагонали) упомянутой области. Прямоугольная штриховая рамка, внутри которой окажутся упорядочиваемые объекты, исчезнет при отпускании кнопки мыши. В результате все требуемые объекты окажутся выделенными квадратными маркерами. Далее щёлкните правой кнопкой мыши по любому из выделенных объектов. Появится контекстное меню. Для *Builder 5* оно показано на левой панели, а для *Builder 6* – правой панели рис. 2.6.
- ❑ В этом меню введите команду *Align* (выравнивание) или *Position/Align*, соответственно для *Builder 5* и *Builder 6*. В результате появится диалоговое окно, показанное на рис. 2.7.

В упомянутом окне имеется две панели, предназначенные для выравнивания объектов по горизонтали (*Horizontal*) и вертикали (*Vertical*). Для расположения объектов вдоль горизонтальной линии применяется их вертикальное выравнивание (перемещение). Поэтому на панели *Horizontal* включите радиокнопку *No change* (не изменять), а на панели *Vertical* – радиокнопку *Tops* (по верхним краям). Если вы желаете, чтобы выравнивание осуществлялось по центру объектов, либо по их нижним краям, то активизируйте соответственно переключатели *Centers* (по центрам) или *Bottoms* (по нижним краям).

Выравнивание объектов формы вдоль вертикальной линии отличается от выравнивания вдоль горизонтальной линии лишь тем, что на панели *Vertical* включается режим *No change*, а на панели *Horizontal* – один из режимов: *Centers*, *Left sides* (по левым краям), *Right sides* (по правым краям).

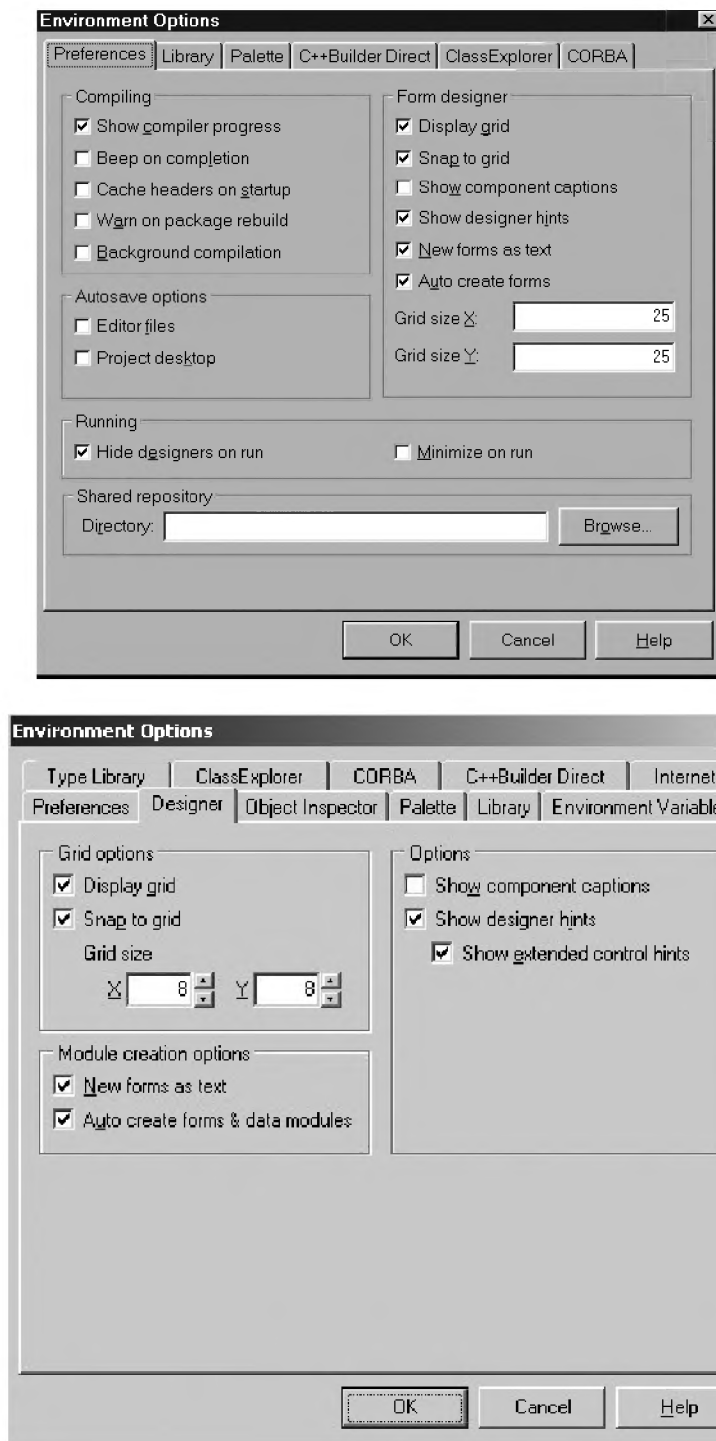


Рис. 2.5. Установка режима
Snap to grid – Привязка к сетке в *Builder 5* (верхняя панель)
и *Builder 6* (нижняя панель)

2.4.5. Выравнивание по сетке

Вершины углов выделенных визуальных объектов можно совместить с ближайшими линиями сетки. Для этого в их контекстном меню (см. рис. 2.6: левая панель для *Builder 5*, правая панель для *Builder 6*) выберете для *Builder 5* пункт *Align to Grid* (выравнивание по сетке), а для *Builder 6* – *Position/Align to Grid* (этот режим выбран по умолчанию).

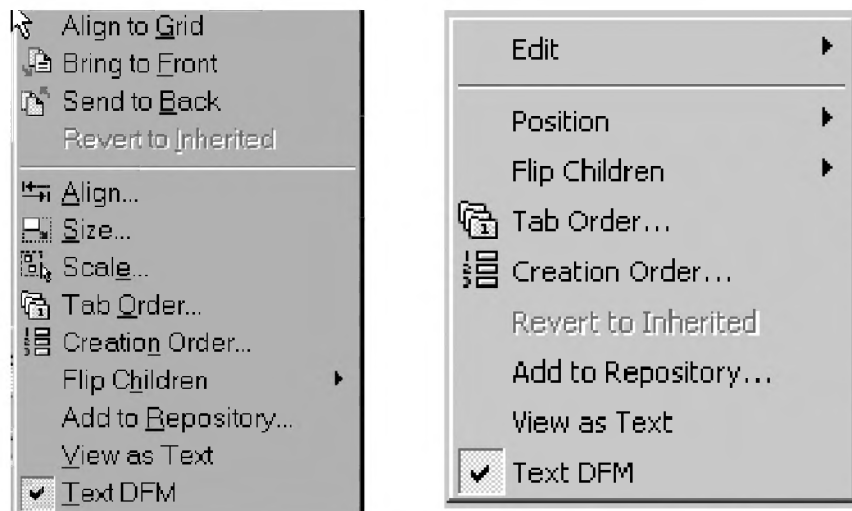


Рис. 2.6. Контекстное меню выделенных объектов.
Левая панель для *Builder 5*, правая панель для *Builder 6*

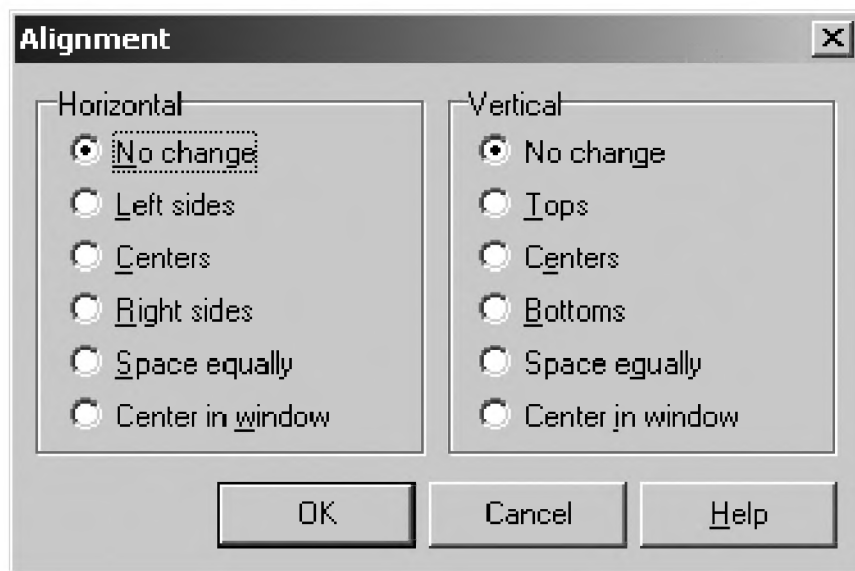


Рис. 2.7. Диалоговое окно режима *Выравнивание*

2.4.6. Уравнивание габаритов

Интерфейс приложения будет выглядеть более привлекательным, если, например, все поля ввода и вывода информации будут иметь одинаковые размеры. Перечислим последовательность операций для уравнивания габаритов группы выделенных объектов:

- ☐ В контекстном меню выделенных объектов (см. рис. 2.6) ввести команду *Size* (размер) или *Position/Size* соответственно для *Builder 5* и *Builder 6*.
- ☐ В появившемся окне *Size* (см. рис.2.8) на панелях *Width* (ширина) и *Height* (высота) выбрать одну из радиокнопок *Shrink to smallest*, *Shrink to largest* (по наименьшему, наибольшему габаритному размеру) или в полях ввода *Width* и *Height* указать значения габаритов в пикселях.
- ☐ После установки *обоих* переключателей нажать кнопку *OK*.

2.4.7. Одинаковые интервалы

Одинаковые интервалы между выделенными объектами устанавливаются при помощи пункта *Align* контекстного меню (см. рис. 2.6). В диалоговом меню этого пункта (см. рис. 2.7) на панели *Horizontal* или *Vertical* следует включить переключатель *Space equally* (распределить равномерно) и нажать кнопку *OK*.

2.4.8. Симметрирование объектов относительно центра

Для симметричного расположения выделенных объектов формы относительно её центра следует вновь вызвать диалоговое окно (см. рис. 2.7) и на панелях *Horizontal* и *Vertical* включить переключатели *Center in window* (выровнять по центру формы), после чего нажать кнопку *OK*.

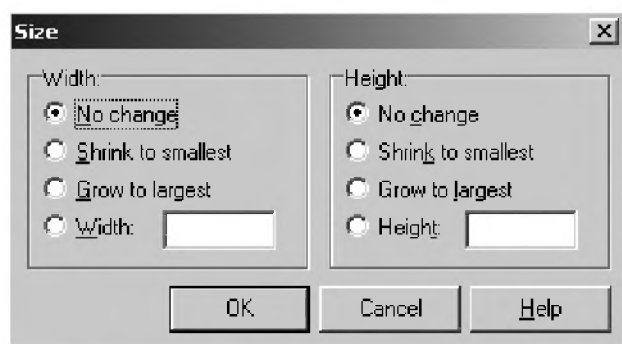


Рис. 2.8. Установка одинаковых размеров группы визуальных объектов

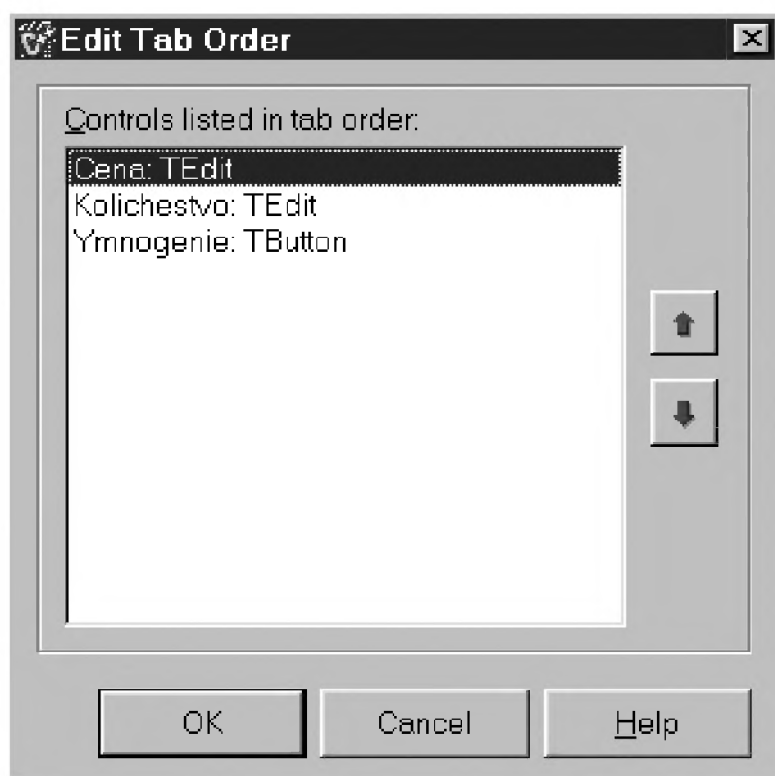


Рис. 2.9. Редактирование порядка перехода фокуса ввода

2.4.9. Порядок перехода между объектами управления

Объектами управления в нашем приложении являются поля ввода *Цена*, *Количество* и кнопка *Умножение*. Во всех стандартных приложениях *Windows* последовательное переключение активности объектов управления (или передача фокуса ввода), помимо использования мыши, осуществляется клавишей *Tab*. Такой способ очень удобен и эргономичен.

При последовательных нажатиях клавиши *Tab* вы обнаружите, что фокус ввода из поля *Цена* переместится в поле *Количество*, а затем на кнопку *Умножение*. Такой порядок перехода достаточно удобен, при конструировании интерфейса в таком порядке вводились эти объекты. Поэтому в приложение *Умножитель* никаких корректировок в порядок перемещения фокуса ввода вносить *не следует*. Однако в вашей практике может возникнуть необходимость в его изменении. Рассмотрим операции, которые для этого необходимо выполнить.

- ☐ Вывести контекстное меню *всех* объектов формы (см. рис. 2.6).
- ☐ Выбрать пункт *Tab Order* (редактирование порядка перехода).
- ☐ В появившемся окне (см. рис. 2.9) при помощи кнопок со стрелками (они нажимаются мышью), изменить положение выделенного объекта в выведенном списке объектов. Этим вы измените предшествующий порядок перехода. Порядок строк списка можно изменять, перетаскивая строки мышью.
- ☐ После установки требуемого порядка перехода нажать кнопку *OK*.

2.5. Подробнее об отладке программы

2.5.1. Предупреждения и подсказки

Для того чтобы в результате компиляции выводились предупреждения и подсказки необходимо для *каждого* отдельного приложения выполнить следующие команды и операции:

- ☐ Ввести команду *Ctrl+Shift+F11* (или *Project/ Options*).
- ☐ В появившемся окне *Project Options* на вкладке *Compiler* (Компилятор) (см. рис. 2.10) в разделе *Warnings* выбрать радиокнопку *All* (все), в разделе *Code optimization* (оптимизация программы) установить радиокнопку *Speed* (оптимизация времени выполнения).
- ☐ Нажать кнопку *OK*.

Теперь в ходе компиляции программы в нижней части окна редактирования кроме информации о найденных ошибках появятся предупреждения и подсказки. Общее число подсказок и предупреждений выводится в окне с результатами компиляции (см. рис. 1. 9).

Для испытания установленных параметров компилятора прокомментируйте последний *try-catch* блок функции *YmnogenieClick*, которые относятся к присваиванию значения переменной *fKolichestvo*. В результате компиляции появится два предупреждения:

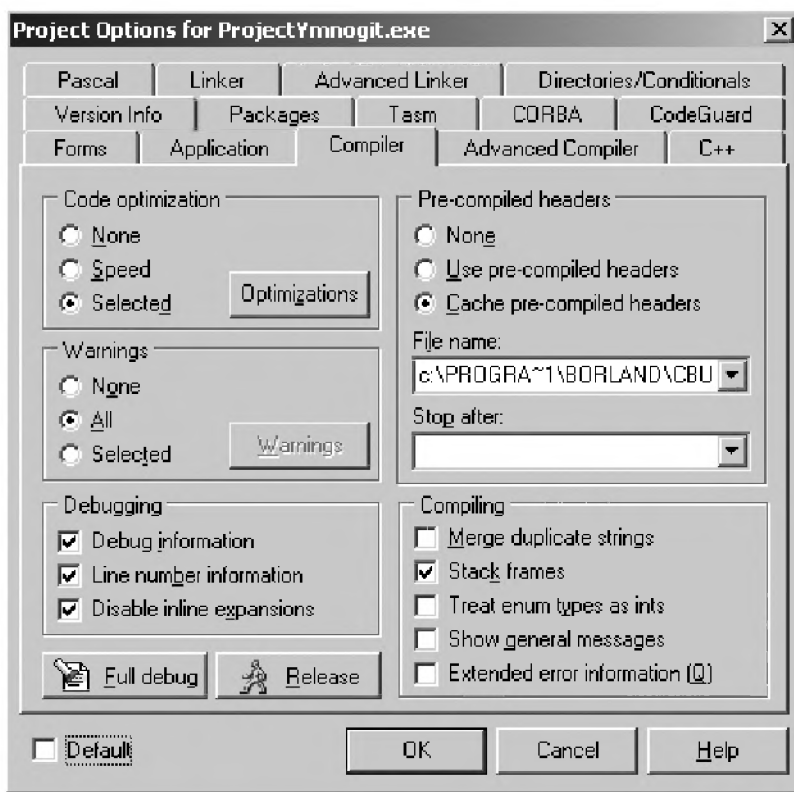


Рис. 2.10. Вывод предупреждений в ходе компиляции программы

- ❑ [C++ Warning] Ymnogitelj.cpp(59): W8013 Possible use of 'fKolichestvo' before definition
- ❑ [C++ Warning] Ymnogitelj.cpp(66): W8057 Parameter 'Sender' is never used

В первом из них компилятор предупреждает об ошибке, которая может наступить в результате того, что переменная *fStoimostj* используется перед присваиванием ей конкретного значения (ведь оператор присваивания был исключён). Об опасности такой ситуации упоминалось в пункте 1.4.3. Дважды щёлкните по этому предупреждению, в результате курсор текстового редактора переместится на соответствующую строку с ошибочным кодом. Эта строка подсвечивается бордовым цветом. Здесь и ниже используются те цвета выделения строк и цвета шрифтов различных объектов программы, которые для соответствующих ситуаций и объектов установлены разработчиком *Builder* (по умолчанию). Режим *По умолчанию* осуществляется выбором константы *default* в окне *Color SpeedSetting*, выводимом командой: *Tools/Editor Options/Color*.

При объявлении переменной, которая нигде не используется (с именем, например, *fLishn_Peremen*) последует предупреждение:

- [C++ Warning] Ymnogitelj.cpp(66): W8080 'fLishn_Peremen' is declared but never used

Второе предупреждение в выведенном списке информирует о том, что параметр *Sender* (отправитель) нигде не используется (*Parameter 'Sender' is never used*). Действительно, в заголовке функции *YmnogenieClick* имеется аргумент-указатель **Sender*, однако внутри этой функции он не применяется. Такие сообщения необходимо просто игнорировать. Вывод подобных очевидных сообще-

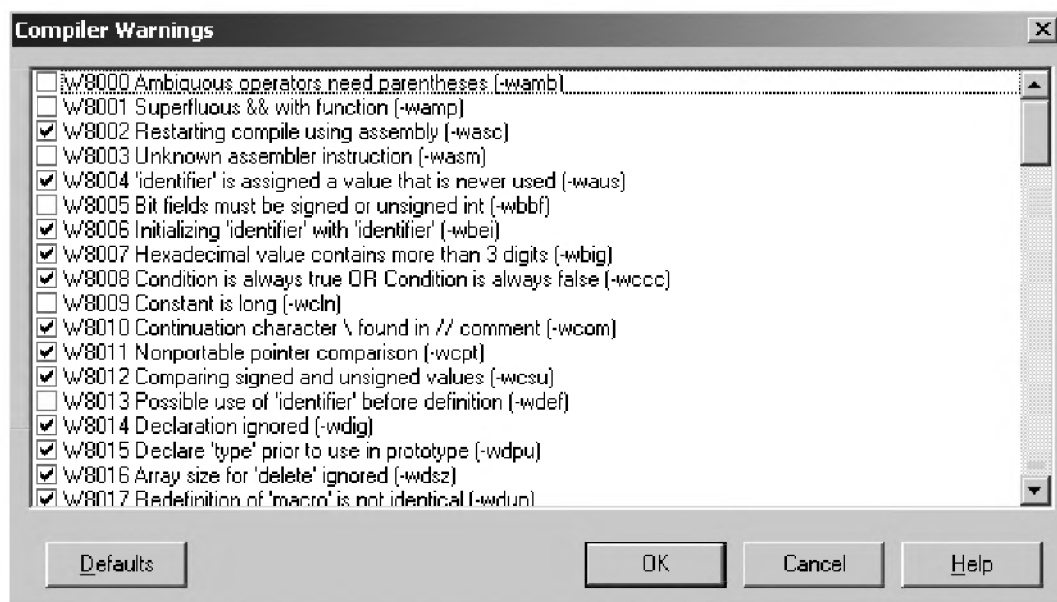


Рис. 2.11. Выбор и исключение предупреждений различного типа

ний можно (но не нужно) отключить. Для этого на вкладке *Compiler* (рис. 2.10) в разделе *Warnings* активизируйте радиокнопку *Selected* (отобранные) и нажмите кнопку *Warnings*. В появившемся списке уберите птички возле ненужных предупреждений (часть списка см. на рис. 2.11). На первых шагах программирования лучше примириться с выводом очевидных предупреждений компилятора и в упомянутом списке оставить активными все пункты, выбранные по умолчанию. Поскольку в противном случае вы можете случайно заблокировать вывод важных предупреждений.

После проверки режима компилятора по выводу предупреждений и подсказок все внесенные ошибки следует устранить. Это можно сделать непосредственным удалением соответствующих знаков многострочного комментария и лишней переменной. Но значительно проще выйти из программы без её сохранения, а затем вновь открыть файл с программой..

2.5.2. Точки остановки и всплывающая подсказка

Для удобства анализа работы программы в нужных её строках можно устанавливать точки остановки (*BreakPoints*). Число таких точек ограничивается лишь числом строк с операторами. Установим, например, точки остановки на последних двух операторах присваивания в функции *YmnogenieClick*. Для установки точки остановки необходимо расположить курсор в требуемой строке и нажать клавишу *F5*. Повторное нажатие этой же функциональной клавиши (на этой же строке) убирает установленную ранее точку остановки. Строка с точкой остановки подсвечивается светло-красным цветом, а слева от строки появляется маркер в виде кружка упомянутого цвета.

Запустите теперь программу на выполнение. Предположим, в первое поле ввода вы занесли «4», а во второе – «5». Поэтому при последовательном наведении курсора на переменные *fCena* и *fKolichestvo* всплывают соответственно подсказки «*fCena*= 4» и «*fKolichestvo*= 5». Для продолжения работы программы

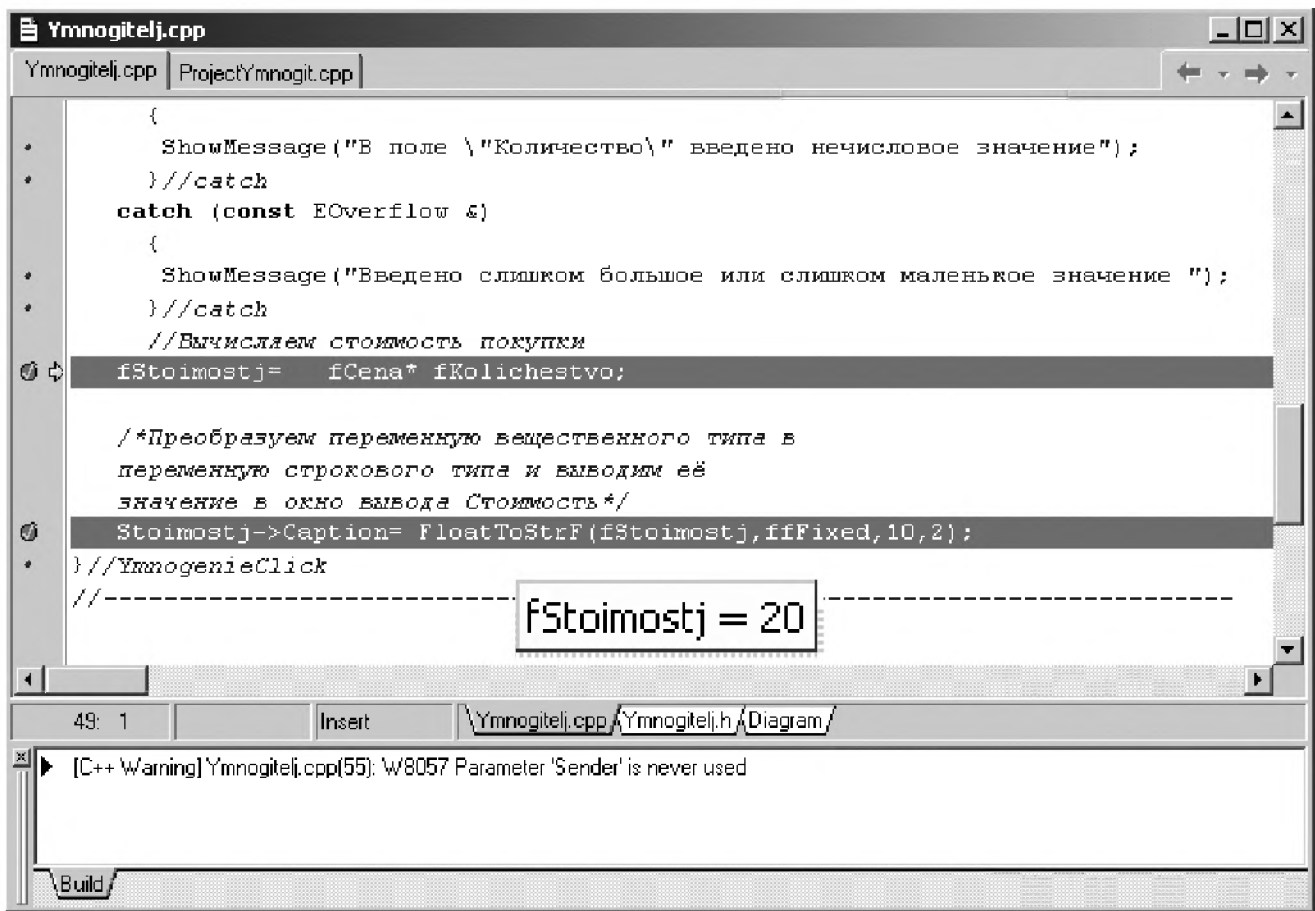


Рис. 2.12. Точки остановки и всплывающая подсказка

вновь нажмите клавишу *F9*. Программа в этом случае будет остановлена уже на второй точке остановки (см. рис. 2.12). Всплывающая подсказка на переменной *fStoimostj* будет такой «*fStoimostj*= 20» ($4 \cdot 5 = 20$).

Обращаем ваше внимание на шрифт различных элементов программы (его цвет и начертание, установленные по умолчанию, можно изменить). Тексты комментариев показаны синим курсивом, синим прямым шрифтом выводятся текстовые константы, а все служебные слова выделены жирным чёрным шрифтом без наклона. Все строки с операторами (но не с комментариями) помечены синими маркерами в виде небольших синих кружков. Маркеры указывают на строки, в которых выполняются *конкретные* действия.

Точки остановки можно устанавливать только в строках с операторами (повторяем, эти строки помечены синими маркерами). Горизонтальная стрелка зелёного цвета указывает строку, на которой работа программы приостановлена, оператор (или операторы) этой строки будут выполняться только на *следующем* шаге программы.

Пусть вас не удивляет, что после запуска программы с установленной в ней точкой (или точками) остановки, синие маркеры стоят перед *всеми* закрывающими фигурными скобками (включая и последнюю). Вместе с тем перед открывающими фигурными скобками маркеров нет. Закрывающая фигурная скобка, как отмечалось ранее, определяет конец логического блока. Поэтому после неё выполняются невидимые для программиста действия, связанные с удалением из оперативной памяти компьютера всех переменных (и объектов), описанных в

данном блоке. После последней фигурной скобки тела функции происходит ещё и целый ряд системных операций *Windows*, завершающих обработку события: нажатие кнопки *Умножение*.

Комбинацией клавиш *Ctrl+F2* корректно завершается работа программы и осуществляется выход в *Редактор Кода*. Уберите все точки останова и при необходимости (если вы сохраняли проект после указанных изменений) сохраните и вновь откомпилируйте проект.

2.5.3. Пошаговое выполнение программы

Эффективным средством поиска ошибок является пошаговое выполнение программы: строка за строкой, оператор за оператором. С использованием всплывающей подсказки при такой неспешной «прогулке» по операторам удобно следить за изменением значений выбранной переменной в теле функции.

Для знакомства с пошаговым режимом компилятора установите точку останова на первом слове *try* функции *YmnogenieClick*. Затем запустите программу, введите допустимые значения в поля ввода (пусть это опять будут числа «4» и «5») и нажмите кнопку *Умножение*. Появится код функции с точкой останова (см. рис. 2.13). Выполнение очередной строки (она выделена синим цветом и слева помечена зелёной стрелкой) осуществляется нажатием клавиши *F8*.

После первого нажатия *F8* в свойстве *Text* объекта *Cena* находится число «4» (см. всплывающую подсказку рис. 2.13), однако переменная *fCena* содержит про-

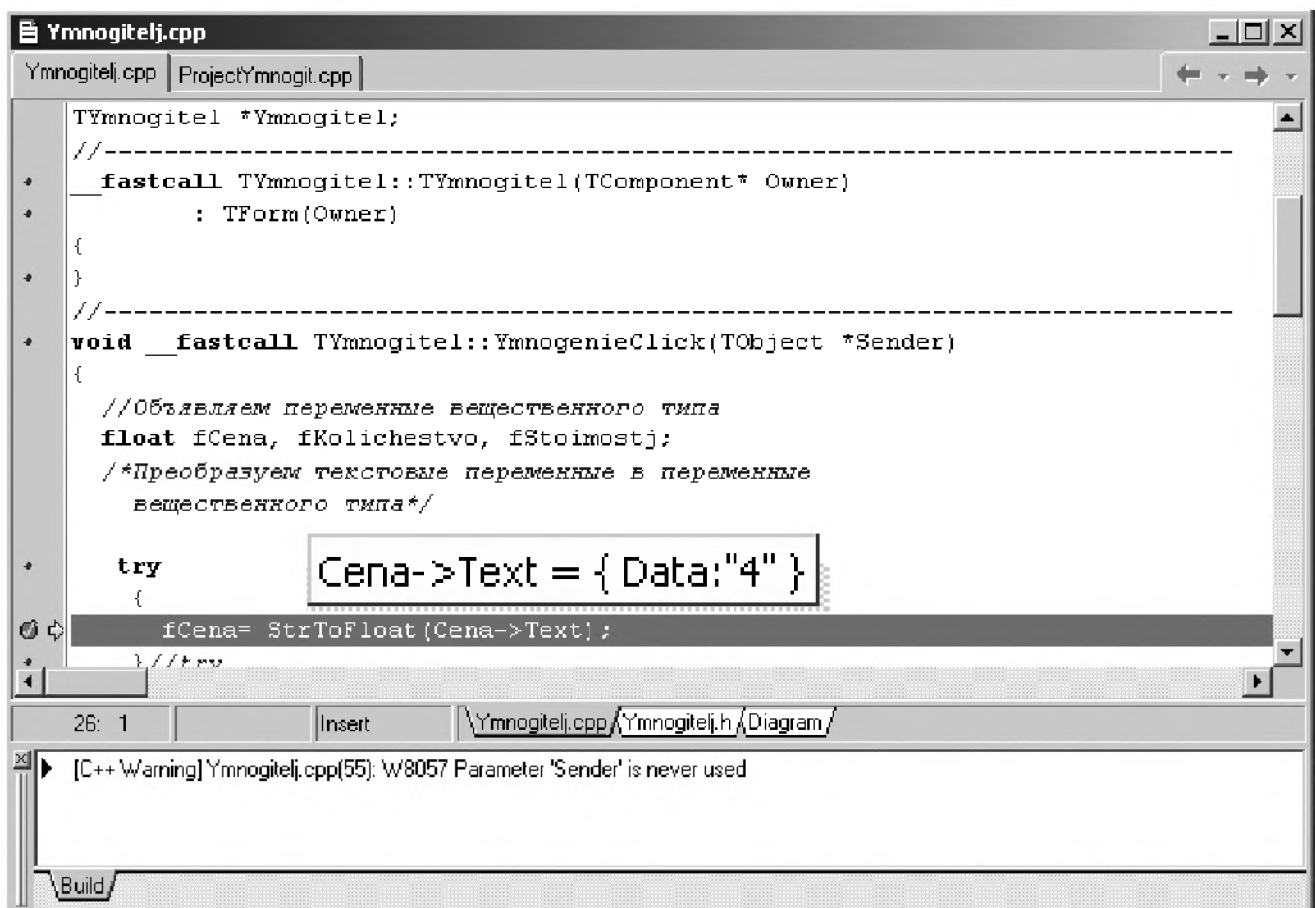


Рис. 2.13. Пошаговое выполнение программы

извольное значение. Это происходит потому, что выделенная строка ещё не выполнялась и поэтому в указанной ячейке памяти находится случайное число, оставшееся в ней от предшествующих операций компьютера. В свойство *Text* помещено значение ещё до начала работы функции *YmnogenieClick* (до нажатия клавиши *Умножение*), поэтому его значение уже определено. Последовательным нажатием *F8* осуществляется аналогичный просмотр всех строк функции.

Если вместо клавиши *F8* нажимать клавишу *F7*, то помимо запуска операторов и функций строки, осуществляется «заход» внутрь функций и пошаговая работа выполняется уже в их теле. Устройство и работа функций подробно изучаются на восьмом уроке, поэтому пока эту команду не используйте.

Пошаговый режим работы ещё называют трассировкой программы. Выйти из режима трассировки можно лишь, завершив его до конца (добравшись до последнего оператора), либо командой *Ctrl+F2* (корректное завершение программы). Вместо упомянутых *машинных* команд трассировки рекомендуем выполнять *ручную* трассировку программы, читая логику ее работы при помощи правил синтаксиса языка программирования. Трассировка «в уме» хорошо структурированных программ часто значительно эффективнее.

2.6. Полноценный исполняемый файл

Режим *отладки* программ *Debug* (устранение ошибок) в *Builder* установлен по умолчанию. В этом режиме в ходе компиляции программы в её исполняемый файл линковщик (редактор связей) включает *не все* коды стандартных функций и классов, использованные в разработанной программе. Эти «пропущенные» (для уменьшения времени компиляции) коды хранятся в *библиотеках времени выполнения* (*Run-Time Libraries*, *RTL*-библиотеки), они внедряются в исполняемый файл только в ходе его запуска. Поэтому такие урезанные *exe*-файлы запускаться *лишь* на компьютерах, на которых установлен *Builder*.

Коды *всех* системных файлов включаются в исполняемый файл режимом *Release* (выпуск, публикация). Он применяется для создания *готового* приложения, его можно установить *только* для *конкретного* проекта. В этом режиме *отключается* весь отладочный инструментарий *IDE*, что повышает быстродействие готового приложения и уменьшает объём его файла. Однако итоговый объём файла всё же увеличивается из-за подключения *RTL*-библиотек. Для переключения на режим *Release* необходимо выполнить следующие действия:

- ☐ Ввести команду *Ctrl+Shift+F11* (или *Project/ Options*).
- ☐ На вкладке *Linker* (см. рис. 2.14) убрать птичку возле режима *Use dynamic RTL* (Использовать динамически подключаемые *RTL*-библиотеки). Для возврата в исходный режим эту птичку необходимо вновь поставить.
- ☐ На вкладке *Packages* (упаковки) сбросить флажок *Build with runtime packages*.
- ☐ На вкладке *Compiler* (см. рис. 2.10) нажать клавишу *Release* (возврат в исходный режим осуществляется клавишей *Full debug*, она находится слева от клавиши *Release*).

- ❑ Нажать кнопку *OK* на окне настройки параметров проекта и выполнить команду горизонтального меню *Project/Build* (Проект/Перестроить). Напомним, что компиляция программы при помощи команды *Ctrl+F9* (или *Project/Make*) затрагивает лишь только те её файлы, которые претерпели изменения. Команда *Build* выполняет компиляцию *всех* файлов проекта без исключения, поэтому даже неизменённые файлы будут перекомпилированы *при новых установках* компилятора.

Режим *Release* в отличие от режима *Full debug* создаёт более быстродействующую программу. Это достигается за счёт устранения всех вспомогательных кодов, предназначенных для её отладки. В режиме *Full debug* исполняемый файл учебной программы урока занимает на диске 27136 байт, а в режиме *Release* – 80 896 байт. Как отмечалось, возрастание кода обусловлено включением в исполняемый файл *RTL*-библиотек.

Если решаемая задача связана, например, со сложной обработкой *большого* объёма информации, то в этом случае быстродействию программы следует уделять *особое* внимание. Повысить быстродействие программы можно настройкой компилятора на конкретный тип процессора:

- ❑ Ввести команду *Ctrl+Shift+F11* (или *Project/Options*).
- ❑ На вкладке *Advanced Compiler* (ускорение компиляции) в разделе *Instruction set* (установка типа процессора) выбрать один из четырёх типов процессора: 386, 486, *Pentium* и *Pentium Pro* (см. рис. 2.15). При процессоре 386 (он установлен по умолчанию) программа может работать на всех компьютерах, однако с минимальным быстродействием, заданным в этом про-

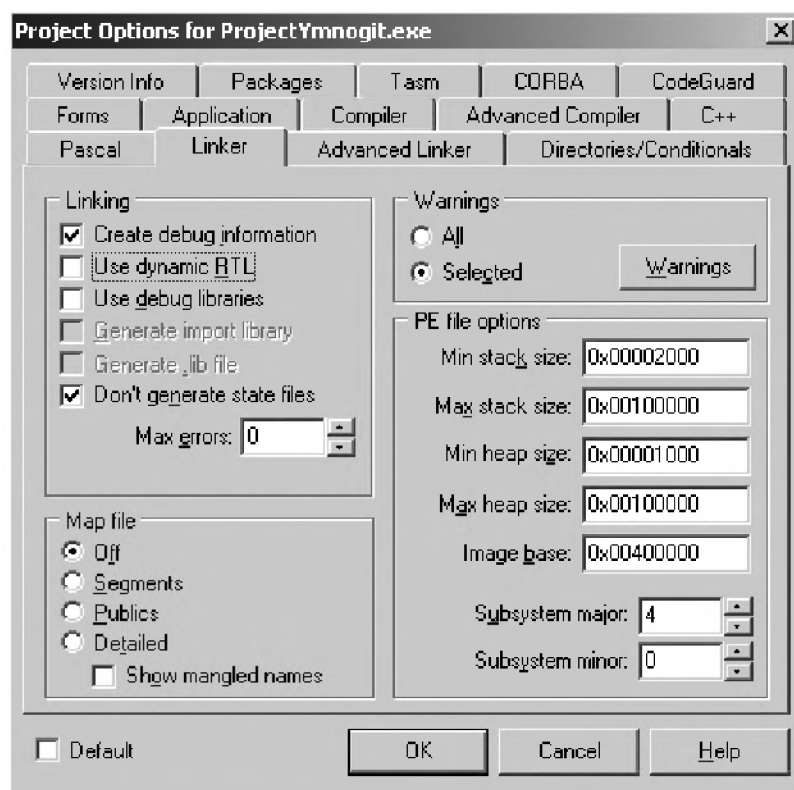


Рис. 2.14. Подключение и отключение *RTL*

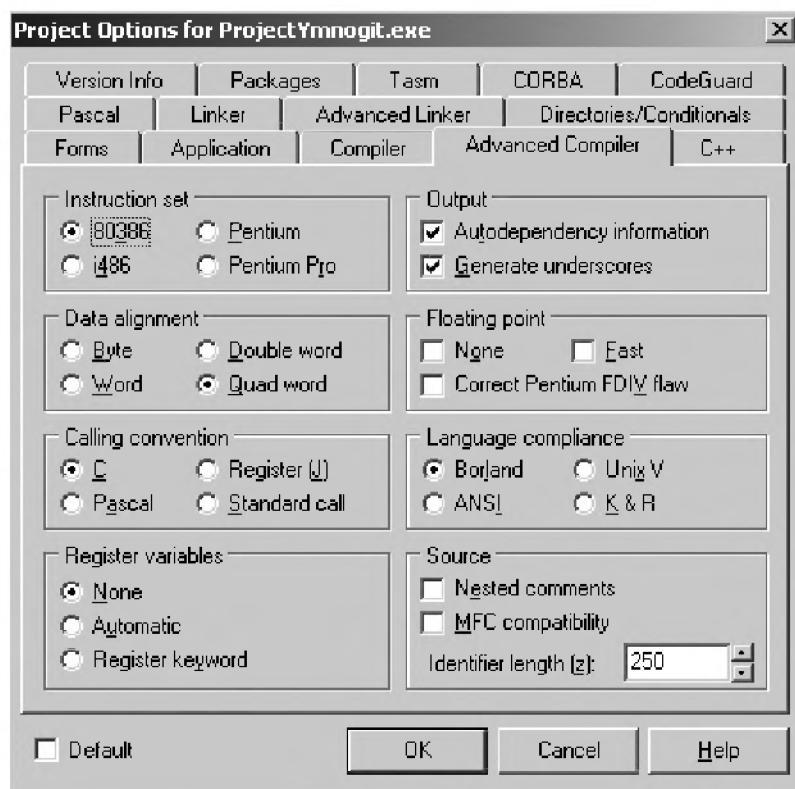


Рис. 2. 15. Выбор типа процессора

цессоре. Если включена радиокнопка более быстродействующего процессора, то в сопроводительных документах необходимо *обязательно* отметить, что на компьютерах ниже этого типа работа приложения не предусмотрена.

Вопросы для самоконтроля

- ☐ Зачем применяются *try-catch*-блоки?
- ☐ Как осуществить уравнивание габаритных размеров объектов интерфейса, выравнивание их местоположения по горизонтали и вертикали, каким образом установить между ними одинаковые интервалы и центрировать относительно краёв формы?
- ☐ В чём заключается наиболее простой способ плавной буксировки и протягивания объектов интерфейса.
- ☐ Каким образом подготовить исполняемый файл проекта, способный запускаться на компьютерах с разным типом процессоров и в случае, когда на них не установлена среда программирования *C++Builder*?
- ☐ Укажите последовательность действий для вывода предупреждений и подсказок в ходе компиляции программы. Зачем они нужны, какие из них можно не принимать во внимание?
- ☐ Перечислите операции по замене пиктограммы проекта.
- ☐ Каким образом можно изменить порядок перехода фокуса ввода (объектов управления) при нажатии клавиши *Tab*?
- ☐ С какой целью применяются точки остановки программы, как они устанавливаются и снимаются?

- ❑ Какие команды (функциональные клавиши) позволяют осуществить пошаговое выполнение программы с заходом и без захода в функции?
- ❑ Назовите команду корректного завершения работы программы.

2.7. Задача для программирования

На первом уроке при самостоятельном программировании разработана программа *Калькулятор*. Предлагается повысить надёжность её работы применением *try-catch*-блоков.

2.8. Вариант программного решения

Здесь приведены только реализации функций по обработке нажатия клавиш.

```
void __fastcall TForm1::YmnogenieClick(TObject *Sender)
{
    float fx, fy, fz;
    try
    {
        fx= StrToFloat(x->Text);
        fy= StrToFloat(y->Text);
    } //try
    catch(const EConvertError &)
    {
        ShowMessage("В одно из полей введены нечисловые данные");
    } //catch
    fz= fx*fy;
    z->Caption= FloatToStr(fz);
} //YmnogenieClick

void __fastcall TForm1::SlogenieClick(TObject *Sender)
{
    float fx, fy, fz;
    try
    {
        fx= StrToFloat(x->Text);
        fy= StrToFloat(y->Text);
    } //try
    catch(const EConvertError &)
    {
        ShowMessage("В одно из полей введены нечисловые данные");
    } //catch
    fz= fx + fy;
    z->Caption= FloatToStr(fz);
} //SlogenieClick

void __fastcall TForm1::VuchitanieClick(TObject *Sender)
{
    float fx, fy, fz;
    try
    {
        fx= StrToFloat(x->Text);
        fy= StrToFloat(y->Text);
    } //try
```



```

    catch(const EConvertError &)
    {
        ShowMessage("В одно из полей введены нечисловые данные");
    } //catch
    fz= fx - fy;
    z->Caption= FloatToStr(fz);
} //VuchitanieClick

void __fastcall TForm1::DelenieClick(TObject *Sender)
{
    float fx, fy, fz;
    try
    {
        fx= StrToFloat(x->Text);
        fy= StrToFloat(y->Text);
    } //try
    catch(const EConvertError &)
    {
        ShowMessage("В одно из полей введены нечисловые данные");
    } //catch
    try
    {
        fz= fx/fy;
    } //try
    catch(const EZeroDivide &)
    {
        ShowMessage("Вы пытаетесь осуществить деление на нуль");
    } //catch
    z->Caption= FloatToStr(fz);
} //DelenieClick

```

Урок 3. Операции, математические функции и операторы выбора

3.1. Использование математических функций и констант

Применение констант и подключение библиотек стандартных функций проиллюстрируем на учебной задаче.

3.1.1. Условие задачи и интерфейс программы *Дальность*

Если не учитывать сопротивление среды, то дальность S полета тела, брошенного под углом α к горизонту с начальной скоростью V , определяется выражением

$$S = V^2 \cdot \sin(2\alpha) / g,$$

где g – ускорение свободного падения. Требуется разработать программу расчёта S по произвольным значениям V и α . Интерфейс программы с полями ввода данных, командной кнопкой и полем для вывода результата расчёта показан на рис. 3.1.

При нажатии кнопки *Расчёт* (её имя *Raschet*) по значениям, введенным в полях *Скорость* (имя V) и *Угол* (имя $Alfa$), вычисляется и выводится в поле *Дальность* (имя S) дальность полёта тела.

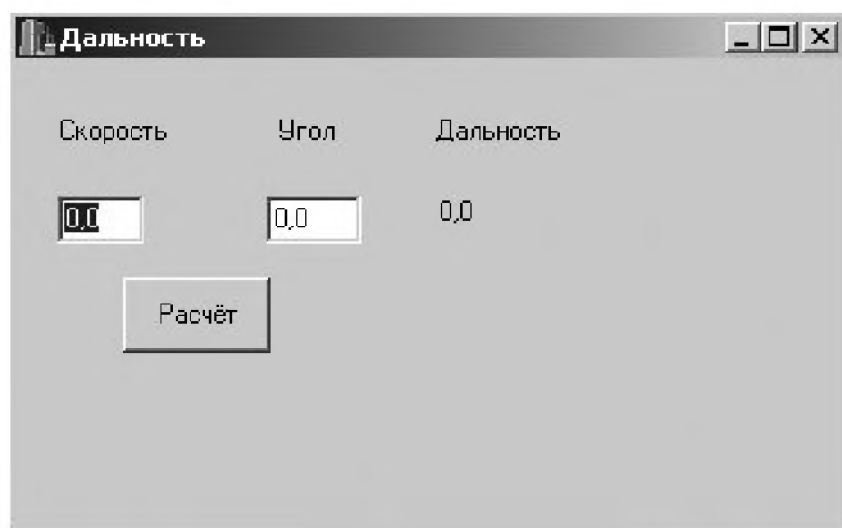


Рис. 3.1. Интерфейс программы *Дальность*

3.1.2. Препроцессорная обработка

Такая обработка выполняет целый *ряд* задач. Рассмотрим вначале только *одну* из них: включение в программу кодов стандартных функций, собранных в тематических библиотеках. В программе *можно* использовать функцию из конкретной библиотеки лишь в том случае, когда эта библиотека *подключена* к программе при помощи директивы препроцессору *include* (включить).

Такие директивы записываются *в самом начале* файлов с реализацией пользовательских функции (в нашем примере это *Dalnostj_Poleta.cpp*), после неё указывается имя подключаемой библиотеки. Директиву препроцессору можно также назвать директивой компилятору.

Визуальные объекты, используемые при разработке интерфейса приложения (форма, поля ввода и вывода, командные кнопки и др.), описаны в заголовочном файле *vcl.h* со стандартной библиотекой визуальных компонентов (*Visual Component Library*). Эта библиотека подключается так:

```
#include <vcl.h>
```

Впереди знака диеза «#» (или знака номера согласно американскому стандарту) пробел не ставится, однако пробел или пробелы допустимы. Впереди и позади слова *include* также могут быть либо отсутствовать пробелы (или пробел). Имя *стандартного* заголовочного файла заключаться в *угловые скобки*. В среде *визуального* программирования эта директива вставляется *автоматически*. По умолчанию также подключается библиотека *stdlib.h*, поскольку её операции применяются при работе визуального проектировщика (см. ниже).

Builder, конечно же, *не знает* какие *другие* объекты и стандартные функции будут использоваться в программе. Поэтому программист должен *самостоятельно* подключить библиотеки с необходимыми функциями. Для этого следует при помощи справочной системы выяснить, в какой библиотеке хранится нужная для решения задачи функция. В справочных статьях, например, для функций *pow* и *sin* указано, что их использование предусматривает включение заголовочного файла *math.h* (от *mathematics* – математика):

```
#include <math.h>
```

Пользовательские заголовочные файлы подсоединяются при помощи *двойных кавычек*:

```
#include "RaschDaln.h"
```

Заголовочный файл *RaschDaln.h*, также как и файл *vcl.h*, подключается к файлу *Dalnostj_Poleta.cpp* *автоматически*, поскольку в нём описывается класс интерфейса, генерируемый *Визуальным Проектировщиком* без участия программиста.

На первых строках файла реализации при помощи директивы *include* подключены упомянутые стандартные и пользовательские заголовочные файлы.

В файле реализации внимательно изучите текст функции *RaschetClick* с её комментарием.

3.1.3. Файл реализации

```
#include <vcl.h>//Подключение библиотеки визуальных компонентов
#include <math.h>//Подключение математической библиотеки
#pragma hdrstop
#include "RaschDaln.h"//Подключение пользовательского
                        //заголовочного файла
#pragma package(smart_init)
#pragma resource "*.dfm"
TDalnostj_Poleta *Dalnostj_Poleta;
__fastcall TDalnostj_Poleta::TDalnostj_Poleta(TComponent* Owner)
    : TForm(Owner)
{
}

void __fastcall TDalnostj_Poleta::RaschetClick(TObject *Sender)
{
    const float g= 9.81;//Ускорение свободного
                        //падения в м/(с*с)
    const float Pi= 3.141593;
    float fV, //Начальная скорость тела в м/с
    fAlfa, //Угол к горизонту вектора начальной
            //скорости тела в градусах
    fAlfaRad, //Значение fAlfa в радианах
    fS; //Дальность полёта тела в м
    fV= StrToFloat(V->Text);
    fAlfa= StrToFloat(Alfa->Text);
    //Преобразование fAlfa в fAlfaRad
    fAlfaRad= fAlfa*Pi/180.0;
    fS= pow(fV,2)*sin(2*fAlfaRad)/g;
    S->Caption= FloatToStrF(fS, ffFixed, 10, 2);
} //RaschetClick
```

3.1.4. Описание программы

После двойного нажатия кнопки *Расчёт* (с именем *Raschet*) *Builder* автоматически создаст заготовку функции *RaschetClick*. Поля ввода *Скорость* и *Угол* имеют соответственно имена *V* и *Alfa*. Полю вывода *Дальность* присвоено имя *S*. В свойство *Text* объектов *V* и *Alfa* записаны нулевые начальные значения. Поэтому свойство *Caption* объекта *S* также содержит нулевое стартовое значение (см. формулу). Имя формы – *Dalnostj_Poleta*, а её заголовок – *Дальность*.

В теле этой функции можно выделить три раздела:

- ☐ описание констант;
- ☐ описание переменных;
- ☐ описание вычислений.

Последние два раздела известны по проекту *Умножитель*. В первом разделе при помощи служебного слова *const* описываются все константы, используемые в формуле для расчёта *S*. Описание каждой константы выполняется так: за ключевым словом *const* через разделительные пробелы указываются соответственно тип и имя константы, при помощи знака присваивания ей назначается конкрет-

ное значение, после которого ставится точка с запятой (впереди неё допустимы пробелы). Число разделительных пробелов не ограничивается: их может быть один, два и более. Однако лишние пробелы ухудшают ясность кода программы.

Можно *все однотипные* константы описать при помощи *одного* слова *const*:

```
const float g= 9.81, Pi= 3.141593;
```

В этом случае описываемые константы разделяются запятыми. При *разнотипных* константах слово *const* и имя типа должно предшествовать *каждой* константе, описания констант разделяются не запятой, а точкой с запятой:

```
const double Pi= 3.141593;  
const float g= 9.81;
```

В последнем примере мы использовали ещё один тип вещественных констант и переменных. Тип *double* (удвоенный) для размещения данных в диапазоне $5,0 \cdot 10^{-324} - 1,7 \cdot 10^{308}$ занимает в памяти 8 байт (в два раза больше, чем у типа *float*). Значения этого типа могут быть как положительными, так и отрицательными.

Рекомендуется константам и переменным программы давать имена, совпадающие или близкие именам соответствующих констант и переменных решаемой задачи. В этом случае программа будет ясной, в большей мере застрахованной от ошибок. Поэтому в функции *RaschetClick* для ускорения свободного падения используется латинская буква *g*, а вот для переменной α выбрано имя *fAlfa*, поскольку греческие буквы в программе недопустимы (напоминаем, буква *f* подсказывает имя типа *float*).

Часто используемые константы описаны в библиотеке математических функций *math.h*. После её подключения в программе можно не описывать, например, константы π , $\pi/2$, $\pi/4$, $1/\pi$, $1/\sqrt{\pi}$ число e ($e=2,7182818\dots$), а вместо них использовать соответственно идентификаторы *M_PI*, *M_PI_2*, *M_PI_4*, *M_1_PI*, *M_1_SQRTPI*, *M_E*.

При определении констант применимы *любые* математические выражения и *ранее* введенные константы. Значения всех констант рассчитываются *на этапе компиляции* программы. В ходе компиляции вычисленные, либо введенные при описании, значения констант заменяют соответствующие им символьные обозначения в теле функции (или в другом логическом блоке их видимости).

В вычислительном разделе после присваивания переменным *fV* и *fAlfa* введенных значений осуществляется пересчет значения угла *fAlfa* из градусной в радианную меру, поскольку в *Builder* аргументы тригонометрических функций задаются в радианах.

Вычисление дальности *fS* происходит в предпоследнем операторе присваивания. Тригонометрическая функция *sin* точно также называется в библиотеке математических функций *Builder*. В указанной библиотеке возведение в степень выполняется функцией *pow(Chislo, Stepenj)*, в которой *Chislo* – возводимое в степень число, а *Stepenj* – степень этого числа. Последний оператор функции *RaschetClick* преобразует вещественный результат вычислений к текстовому типу и выводит его в свойстве *Text* объекта *S*.

3.2. Основные стандартные математические функции

В заголовочном файле *math.h* находится ряд часто используемых математических функций. Ниже познакомимся с теми из них, которые применяются в дальнейших примерах. Материал настоящего и последующего разделов носит справочный характер, при первом чтении его можно лишь бегло *просмотреть*.

Перед каждой функцией указывается тип значения, который она возвращает, а впереди аргумента (или аргументов) приводится его (или их) разрешённый тип.

- ❑ *double sin(double x), long double sinl(long double x)* – функции синуса. Отличие второй функции от первой лишь в типе её аргумента и возвращаемого значения. Вещественные числа типа *long double* занимают 10 байт памяти, числа этого типа находятся в диапазоне: $3,4 \cdot 10^{-4932} - 1,1 \cdot 10^{4932}$, могут быть положительными и отрицательными. Слово *long* (длинный), стоящее впереди некоторых основных типов, называется *модификатором типа*. В других типах (см. ниже) модификатором может быть также слово *unsigned* (беззнаковый).
- ❑ *double cos(double x), long double cosl(long double x)* – функции косинуса.
- ❑ *double tan(double x), long double tanl(long double x)* – функции тангенса.
- ❑ *double asin(double x), long double asinl(long double x)* – функции арксинуса.
- ❑ *double acos(double x), long double acosl(long double x)* – функции арккосинуса.
- ❑ *double atan(double x), long double atanl(long double x)* – функции арктангенса. Функции *asin()*, *acos()*, *atan()* возвращают значения в пределах от $-\pi/2$ до $\pi/2$.
- ❑ *double pow(double x, double y), long double powl(long double x, long double y)* – функции возвращают значение x^y .
- ❑ *double sqrt(double x), long double sqrtl(long double x)* – функции возвращают положительное значение квадратного корня, их аргумент должен быть положительным.
- ❑ *double log(double x), long double logl(long double x)* – возвращают *натуральный* логарифм x , аргумент должен быть положительным, математическая запись функции – $\ln(x)$.
- ❑ *double floor(double x), long double floorl(long double x)* – находят наибольшее целое, не превосходящее значение x (округление вниз, см. рис. 3.2). Примеры: $\text{floor}(3.14) = 3$, $\text{floor}(-3.14) = -4$.
- ❑ *double ceil(double x), long double ceill(long double x)* – наименьшее целое, не меньшее x (округление вверх, см. рис. 3.3). Примеры: $\text{ceil}(3.14) = 4$, $\text{ceil}(-3.14) = -3$.

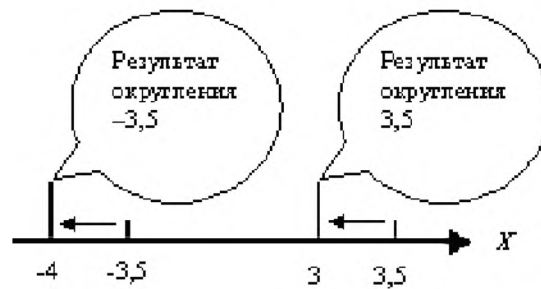


Рис. 3.2. Примеры округления (вниз) вещественных значений при помощи функции *floor()*.



Рис. 3.3. Примеры округления (вверх) вещественных значений при помощи функции *ceil()*.

- ❑ *double fmod(double x, double y), long double fmodl(long double x, long double y)* – возвращают остаток от деления нацело x на y . Пример: $fmod(5, 3) = 2$.
- ❑ *double fabs(double x), long double fabsl(long double x)* – возвращают абсолютное вещественное значение вещественно аргумента x . Пример: $fabs(-3.14) = 3.14$.
- ❑ *long labs(long int x)* – возвращает абсолютное целочисленное значение целого аргумента x . Пример: $labs(-20022006) = 20022006$.

Для размещения в памяти компьютера целых чисел типа *long int* требуется 8 байт, числа этого типа находятся в интервале от $-2\,147\,483\,648$ до $2\,147\,483\,647$.

Заголовочный файл *stdlib.h* (от **standard** – стандарт) содержит различные функции и операции, среди них имеются и математические функции. Эту библиотеку можно не подключать к проекту, поскольку она присоединяется к *любой* разрабатываемой программе *автоматически* ввиду того, что в ней описаны операции для работы с динамической памятью (*new*, *delete*), применяемые в работе с визуальными объектами. Упомянутые операции широко используются в 10 и 14 уроках. Сейчас же приведём несколько функций из этой библиотеки, необходимые на *ближайших* уроках.

- ❑ *int abs(int x)* – возвращает абсолютное целочисленное значение целого аргумента x . Пример: $abs(-2006) = 2006$.

Целые числа типа *int* (от *integer* – целое число) требуют для размещения 4 байта, числа находятся в том же интервале, что и в типах *long* и *long int*: от – 2 147 483 648 до 2 147 483 647.

❑ *int rand(void)* – генерирует псевдо-случайные числа, равномерно распределённые в диапазоне от 0 до 32 767.

❑ *int random(int chislo)* – генерирует псевдо-случайные числа, равномерно распределённые в диапазоне от 0 до (*chislo*–1). Здесь *chislo* – целочисленный аргумент функции.

❑ *void randomize(void)* – если эту функцию применить перед функциями *rand* и *random*, то генерируемые ими числа будут не псевдо-случайными, а случайными, они не будут повторяться при повторных запусках программы.

3.3. Операции и порядок их выполнения

Операции производятся с использованием операндов. *Операнды* – это всё то, что стоит между математическими знаками в выражениях. Их полное определение можно сформулировать так: операндами называют объекты операций, реализуемые компьютером в ходе выполнения вычислений. Таким образом, операндами являются идентификаторы констант, переменных и функций.

3.3.1. Операции отношения

Операции отношения предназначены для сравнения двух величин или операндов. Результат сравнения имеет логический тип *bool*. Операции отношения возвращают *true* (или число, *отличное от нуля*), если указанное соотношение операндов выполняется, и *false* (или 0), если соотношение не выполняется. Определены следующие операции отношения:

`==` – равно,

`!=` – не равно,

`<` – меньше,

`>` – больше,

`<=` – меньше или равно,

`>=` – больше или равно.

3.3.2. Круглые скобки

Круглые скобки используются для заключения в них части выражения, значение которой необходимо выполнить в первую очередь. В выражении можно использовать *любое* количество круглых скобок. При этом число открывающих и количество закрывающих круглых скобок должны *совпадать*. Их можно ставить и просто для удобства чтения выражения (чтобы код программы стал более ясным для быстрого восприятия).

Операция	Действие	Выражение	A	B	Результат
!	Логическое отрицание	!A	true false		false true
&&	Логическое И	A &&B	true true false false	true false true false	true false false false
	Логическое ИЛИ	A B	true false true true	true true false true	false false true false

3.3.4. Арифметические операции

Если знак операции стоит между двумя операндами, то такая операция называется бинарной. Бинарные арифметические операции приведены в таблице 3.2.

Таблица 3.2. Бинарные арифметические операции

Обозначение	Название	Пример
+	сложение	$x + y$
–	вычитание	$x - y$
*	умножение	$x * y$
/	деление	x / y
%	остаток целочисленного деления	$5 \% 2 == 1$

Операция называется *унарной*, если знак операции стоит только возле *одного* операнда. Унарные арифметические операции описаны в табл. 3.3.

Таблица 3.3. Унарные арифметические операции

Обозначение	Название	Пример
+	подтверждение знака или унарный плюс	+52
–	изменение знака или унарный минус	–49
++	инкремент	iPeremen++; ++iPeremen
--	декремент	iPeremen--; --iPeremen

Во *всех* математических выражениях операнды могут быть *любых* числовых типов. При разных типах операндов перед выполнением арифметического действия операнд с менее точным типом приводится к более точному, происходит *уравнивание* типов операндов по операнду более *старшего* типа (имеющего больше разрядов и диапазон допустимых значений). Таким образом, результат имеет более *старший* тип. Однако в операции вычисления остатка от целочисленного деления «%» оба операнда должны иметь *целочисленный* тип.

В операциях деления и вычисления целочисленного остатка второй операнд *не может* равняться нулю. Знак результата вычисления целочисленного деления *совпадает* со знаком *первого* операнда. Особое внимание следует уделять ситуации, когда оба операнда в операции деления имеют целочисленный тип (или просто целые значения), а результат деления является не целым числом. Поскольку в этом случае для операции деления используются следующие правила:

1. Если первый и второй операнды имеют *одинаковые* знаки, то результат операции деления – наибольшее целое, меньшее истинного результата деления.
2. Если первый и второй операнды имеют *разные* знаки, то результат операции деления – наименьшее целое, большее истинного результата деления.

Таким образом, округление всегда осуществляется *по направлению к нулю* (см. рис. 3.6).

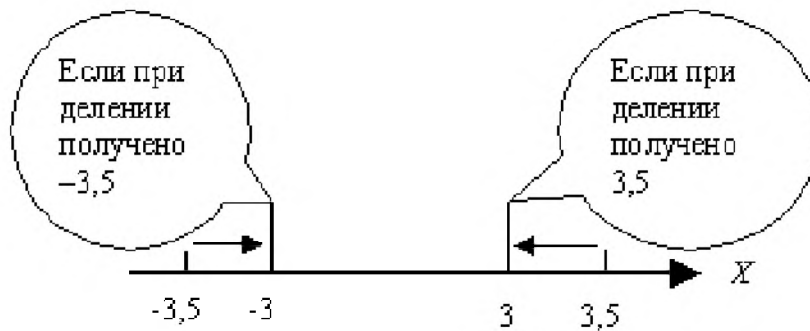


Рис. 3.6. Примеры округления (по направлению к нулю) при делении целых чисел или целочисленных переменных.

Унарные операции инкремента ($++$) и декремента ($--$) применимы *только к переменным*, они сводятся соответственно к увеличению и уменьшению операнда-переменной на единицу (подробно рассматриваются на следующем уроке). Операции инкремента и декремента выполняются быстрее, чем соответствующие операторы присваивания (подробнее см. п. 3.3.5).

3.3.5. Операции присваивания

Если тип операндов по разные стороны оператора присваивания отличается друг от друга, то в результате присваивания тип результата *преобразуется* к типу *левого* операнда. Пример: `int ix = 3.5`; (в `ix` записывается число 3). Поэтому, если тип левого операнда *младше*, чем тип результирующего выражения, то возможна потеря точности или вообще неправильный результат.

Имеется ряд *составных* операторов присваивания, широко используемых в подавляющем большинстве программ разработанных на C++. Например, дополнительно к оператору присваивания `X = X + Y` введена компактная его форма: `X += Y`. В обоих операторах присваивания к значению, расположенному в переменной `X`, добавляется значение, хранящееся в переменной `Y`, после чего результирующая сумма помещается в переменной `X`. Аналогично определяются и другие операции присваивания: `<-=`, `<*=`, `</=`, `<%=`. В частности, `X %= Y` эквивалентно `X = X % Y`. При записи составных операций присваивания символ операции и обычный знак присваивания записываются *слитно*, без пробела. Использование составных операций присваивания ускоряет выполнение соответствующих вычислений.

3.3.6. Порядок вычислений

В первую очередь вычисляются выражения, заключенные в круглых скобках. Для вложенных друг в друга двух пар круглых скобок с выражениями производится расчёт вначале внутреннего выражения, а затем внешнего. Приоритеты действий, исполняемых при вычислении выражений, приведены в таблице 3.4.

Все операции отношения, операции круглые скобки `()`, умножение `*`, деление `/`, определение остатка целочисленного деления `%`, сложение `+` и вычитание `-` выполняются *слева направо*. Все унарные операции и операции присваивания исполняются *справа налево*.

Таблица 3.4. Порядок вычислений

Приоритет операций	Тип действия
1	Вычисление в круглых скобках
2	Вычисление значений функций
3	!, унарный +, унарный -, инкремент ++, декремент --
4	Умножение *, деление /, %
5	Сложение +, вычитание -
6	Операции отношения <, <=, >, >=
7	Операции отношения =, !=
8	Операция отношения &&
9	Операция отношения
10	Операции присваивания =, *=, /=, %=, +=, -=

3.4. Условный оператор *if*

Условный оператор *if* предназначен для *ветвления* работы программы. Познакомимся с ним вначале в общем виде:

```
if (условие)
{
    //Группа операторов 1, выполняется, если
    //соблюдается условие
}
[else
{
    //Группа операторов 2, выполняется, если
    //нарушается условие
}]
```

После служебного слова *if* (если) в круглых скобках стоит условное выражение. Это выражение может выполняться (тогда его значение равно *true*), либо не выполняться (*false*). Напоминаем, помимо приведенных служебных слов, результаты логических выражений могут оцениваться числами. Ложные результаты равны *нулю*, истинные – любому числу, *не равному нулю*.

Если в приведенном выше коде *условие* истинно (выполняется) то исполняется только «Группа операторов 1», при этом «Группа операторов 2», стоящая после ключевого слова *else* (иначе), пропускается. Если же *условие* ложно, то под-

Квадратными скобками отмечены те составляющие условного оператора, которые могут отсутствовать. Таким образом, разрешается использовать *сокращенную* форму условного оператора:

Если значение условия выполняется, то исполняются операторы, заключённые в последующем логическом блоке, в противном случае весь условный оператор *пропускается*. Если в каждом логическом блоке условного оператора *if_else* находится только *по одному* оператору, то в этом случае фигурные скобки для выделения таких блоков *можно* не использовать.

```
void __fastcall TForm1::DelenieClick(TObject *Sender)
{
    float fx, fy, fz;
    try
    {
        fx= StrToFloat(x->Text);
        fy= StrToFloat(y->Text);
    } //try
    catch(const EConvertError &)
    {
        ShowMessage("В одно из полей введены нечисловые данные");
    } //catch
    if (fy== 0)
    {
        ShowMessage("Вы пытаетесь осуществить деление на нуль");
        Abort(); //Генерация "молчаливого" исключения
    } //if
    else
        fz= fx/fy;
    z->Caption= FloatToStr(fz);
} //DelenieClick
```

Первая ошибка: логическое выражение *проверка на равенство* ($fy == 0$) применяется для *вещественных* значений. Проверка на равенство для вещественных значений *недопустима*. В программе необходимо указывать *интервал*, при попадании в который результат сравнения считается положительным (*true*). Такое

требование обусловлено тем, что из-за ошибок округления и различных вариантов представления *вещественных* чисел (например, 2.0 и 1.9999...) логическое выражение *проверка на равенство* (предназначенное *только* для чисел и переменных *целочисленных* типов (*int*, *long int*)) может *не выполняться*.

Вторая ошибка: при недопустимых введенных данных после разрушения исключительной ситуации (нажатия кнопки *ОК* всплывающего окна с соответствующим предупреждением) в поле результата выводится значение для предшествующего вычисления, что может быть неправильно понято пользователем приложения.

Для устранения указанных недостатков этот фрагмент функции следует представить, например, в таком виде:

```
if (fy<= 1E-37 && fy>= -1E-37)
{
    ShowMessage("Вы пытаетесь осуществить деление на нуль");
    z->Caption=""; //или z->Caption="Ошибка";
    Abort(); //Генерация "молчаливого" исключения
} //if
else
    fz= fx/fy;
z->Caption= FloatToStr(fz);
```

После вывода предупреждения о том, что частное равно нулю (величина, которая может попасть в приведенный интервал, практически равна нулю) и нажатия кнопки *ОК* функция *Abort()* генерирует молчаливое исключение *EAbort*, в результате которого произойдет корректное завершение вычислений с неправильными данными (см. п. 2.2.2). Перед таким экстренным завершением программы в поле результата (где осталось значение от *предшествующего* вычисления) записывается пустая строка (предшествующее значение стирается) или выводится слово *Ошибка*.

Количество применяемых операторов *if_else* и краткой его формы *if* в программах не ограничивается. Эти операторы могут быть также вложенными друг в друга:

```
if (условие 1)
    Оператор_1;
else
    if (условие 2)
        Оператор_2;
    else
        if (условие 3)
            Оператор_3;
        else
            ShowMessage("Ky-Ky");
```

Сообщение, приведенное в функции *ShowMessage*, появится на экране *только* в результате невыполнения *всех трёх* условий.

Каждый из вложенных условных операторов может не иметь разделителя *else*, то есть использоваться в сокращенном виде. В этом случае возникает очевидный вопрос: к какому *if* относится выбранный разделитель *else*? Ответ определяется правилом: каждый разделитель *else* принадлежит *ближайшему сверху* опера-

тору *if*. Руководствуйтесь им для понимания логики вложенных операторов *if*.

В качестве примера изучите следующий фрагмент программы, удостоверьтесь в том, что на экран будет выведено лишь *последнее* сообщение.

```
...
const int a = 1, b = 2, c = 3;
if (a < b)
    if (a < c)
        if (b > c)
            ShowMessage("Будет ли выведено это сообщение?");
        else
            ShowMessage("Я отношусь к ближайшему if");
...
```

Применением фигурных скобок всегда можно нарушить упомянутое правило:

```
...
const int a = 1, b = 2, c = 3;
if (a < b)
{
    if (a < c)
        ShowMessage("Я покажусь на экране!");
}
else
    ShowMessage("Меня не выведут на экран.");
```

В приведенных фрагментах логика работы программы легко прослеживается в результате применения отступов, очень помогающих разобраться в том, какие операторы в какой логический блок входят. Повторяем, что такое структурирование кода программы является одним из важных пунктов хорошего стиля программирования, неперенным стандартом разработки *всех* приложений.

Ниже приведена функция *Raschet_ifClick*, она является вариантом функции *RaschetfClick*, в котором при помощи оператора ветвления *if_else* проверяется правильность введенных данных.

```
void __fastcall TDalnostj_Poleta::Raschet_ifClick(TObject *Sender)
{
    const float g= 9.81, Pi= 3.141593;
    float fV, fAlfa, fAlfaRad, fS;
    //fV Начальная скорость тела в м/с
    //fAlfa - угол к горизонту вектора начальной
    //скорости тела в градусах
    //fAlfaRad - значение fAlfa в радианах
    //fS дальность полёта тела в м

    fV= StrToFloat(V->Text);
    fAlfa= StrToFloat(Alfa->Text);
    if (fV>= 0 && fAlfa>= 0 && fAlfa<= 180)
    {
        fAlfaRad= fAlfa*Pi/180.0;
        fS= pow(fV, 2.0)*sin(2*fAlfaRad)/g;
        S->Caption= FloatToStrF(fS, ffFixed, 10, 2);
    } //if
    else
```

```
{
    S->Caption= ""; //Стирается предшествующий результат
    ShowMessage("Введены недопустимые значения");
} //else
} //Raschet_ifClick
```

Если вы ввели недопустимые значения для скорости или угла, то будет стёрт предшествующий результат и в специальном окне на экране появится сообщение: *Введены недопустимые значения*. Это произойдёт в результате работы логического блока, который стоит за служебным словом *else*, а логический блок за ключевым словом *if* в этом случае будет пропущен. После нажатия клавиши *OK* (или клавиши *Enter*) на экране вновь появится интерфейс приложения. В полях ввода исходных данных будут находиться введенные вами значения, поэтому легко определить, какое из них оказалось недопустимым. При допустимых же значениях исходных величин выполняется логический блок, расположенный после зарезервированного слова *if*, логический блок за словом *else* пропускается.

Для закрепления рассмотренных ранее приёмов, предназначенных для контроля вводимых данных, дайте ответы на следующие вопросы:

1. Какие изменения следует внести в текст функции *Raschet_ifClick* для того, чтобы информировать пользователя приложения о вводе им нечисловых значений в конкретном поле ввода?
2. Как осуществить вывод слова *Ошибка* в поле *S* всякий раз, когда появляется сообщение о неверных исходных данных?

3.5. Оператор множественного выбора *switch*

При помощи *совокупности* операторов *if* можно осуществить множественное ветвление программы в зависимости от условий, складывающихся в процессе её выполнения. Поэтому такой оператор позволяет конструировать программы с большим количеством условий. Однако, когда число условий выбора (альтернатив) превышает три, более *удобным* является применение оператора множественного выбора *switch* (переключатель).

Проиллюстрируем его возможности при модернизации программы *Калькулятор*. На интерфейсе программы имелись четыре кнопки для осуществления одного из арифметических действий над числами, набранными в двух полях ввода (см. рис. 1.11). В новом варианте приложения, назовём его *Вычислитель* (имя формы *Vuchislitelj*), будет только одна кнопка *Пуск*. Её нажатие производится после ввода значений *X*, *Y* и номера одной из арифметических операций, перечисление которых (или меню) приведено на интерфейсе (см. рис. 3.7). Для выбора операции необходимо в поле ввода *Действие* ввести одну из цифр от 1 до 4. Заголовки *Действие* и номера арифметических действий располагаются на трёх строках объекта типа *TLabel*. Для возможности ввода в экземпляре объекта типа *TLabel* многострочного текста в его свойстве *WordWrap* (перенос строки) значение *false* следует изменить на *true*.

Ниже приведена функция *PyskClick*, предназначенная для обработки сообще-

ния, вырабатываемого в результате нажатия кнопки *Пуск* (её имя *Pusk*). В упомянутой функции введена ещё одна переменная *iOperacija*, она имеет целочисленный тип *int*. Эта переменная получает числовое значение, которое в текстовом виде вводится в свойстве *Text* поля ввода *Действие* (его имя *Dejstvie*). Для преобразования текстового значения числа в числовой целочисленный типа *int* применяется функция *StrToInt* (строка в целое число).

```
void __fastcall TVuchislitelj::PyskClick(TObject *Sender)
{
    float fx, fy, fz;
    int iOperacija;
    try
    {
        fx= StrToFloat(x->Text);
        fy= StrToFloat(y->Text);
        iOperacija= StrToInt(Dejstvie->Text);
    } //try
    catch(const EConvertError &)
    {
        ShowMessage("В одно из полей введены нечисловые данные");
        z->Caption= "";
        Abort(); //Генерация "молчаливого" исключения
    } //catch

    switch (iOperacija)
    {
        case 1 : fz= fx + fy;
                break;
        case 2 : fz= fx - fy;
                break;
        case 3 : fz= fx*fy;
                break;
        case 4 : if (fy<=1E-37 && fy>=-1E-37)
                    {
                        ShowMessage("Вы пытаетесь осуществить деление на нуль");
                        z->Caption= "";

```

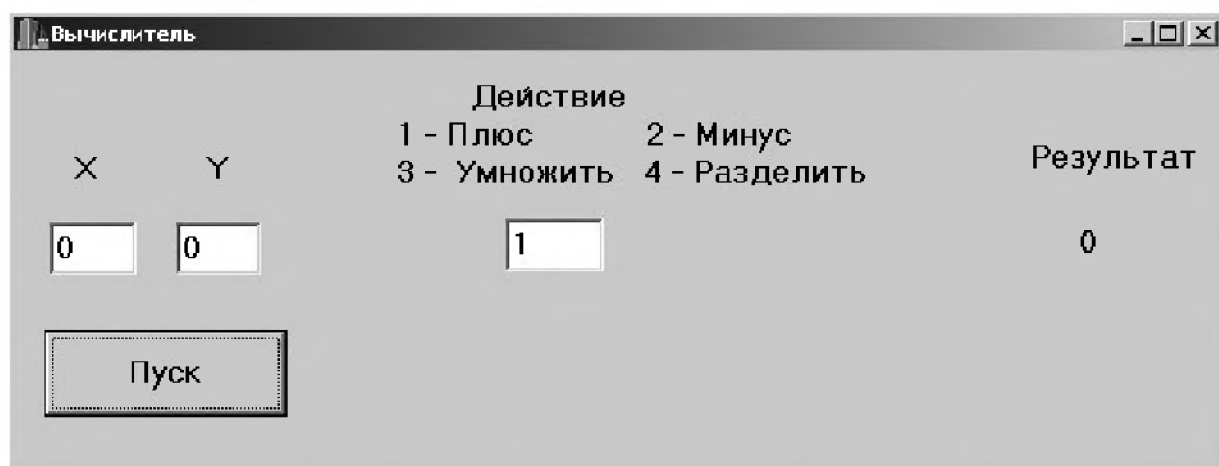


Рис. 3. 7. Интерфейс программы *Вычислитель*

```

        Abort();
    } //if
else
{
    fz= fx/fy;
    break;
} //else

default : ShowMessage("Введен недопустимый знак");
} //switch

z->Caption= FloatToStrF(fz, ffFixed, 10, 3);
} //PyskClick

```

Если пользователь приложения введёт число вне диапазона 1 – 4, то оператор *switch* передаст управление программой на логический блок, расположенный за меткой *default* (не выполнение условий). В нашем примере в этом логическом блоке находится всего *лишь один* оператор (*ShowMessage("Введен недопустимый знак");*), поэтому он не выделен фигурными скобками. При вводе в поле *Действие* нечислового значения появится соответствующее предупреждение.

Допустимые значения (числа от 1 до 4) осуществляют выбор логического блока, расположенного за меткой *case* (вариант) с числом, которое совпадает со значением, введенным в поле *Действие*.

После первых трёх меток помимо оператора присваивания (в котором реализуется операция сложения, вычитания либо умножения) стоит также оператор *break* (прерывать, нарушать). Он прерывает выполнение оператора *switch* в результате чего управление передаются следующему за ним оператору. В четвёртой метке *case*, после цифры «4» и двоеточия отслеживается ситуация деления на нуль и выполняется деление введенных чисел.

Таким образом, *четыре* прежние функции выполнения арифметических действий заменяются *одной* функцией, что приводит к большей компактности и ясности программы. Рассмотрим теперь возможности оператора *switch* более подробно. В общем виде этот оператор записывается так:

```

switch (selektor)
{
    case Znachenie_1 : Operator_1;
                     [break;]
...
    case Znachenie_n : Operator_n;
                     [break;]
[default : Operator_n1;]
}

```

Здесь, как и в операторе *if*, необязательные операторы помещены в квадратные скобки. Тип переменной *selektor* должен быть порядковым: целочисленным, литерным или перечисляемым (о последних типах будет рассказано на 12 и 13 уроках). Если значение переменной-селектора *selektor* не соответствует ни одному из перечисленных после меток *case*, то выполняется оператор (или блок операторов), следующий за меткой *default*. Если же эта необязательная метка отсутствует, и не нашлось метки *case* со значением, соответствующим селектору, то

переключатель пропускается (игнорируется).

При совпадении переменной *selektor* с одним из значений после метки *case* выполняется оператор (или блок операторов), расположенный между двоеточием и оператором *break*. Назначение оператора *break* состоит в том, чтобы передать управление за переключатель *switch*. Таким образом осуществляется запуск только *одного* логического блока, помещённого за конкретной меткой.

Если не использовать необязательные операторы *break*, то вначале выполнится оператор (или группа операторов), находящийся после метки со значением, которое совпало с переменной *selektor*. А далее будут *последовательно* выполняться *все* операторы и блоки, расположенные после *всех* остальных меток, *включая* и метку *default*. Такой режим работы переключателя требуется очень редко, поэтому обычно прибегают к использованию оператора *break*, передающего управление оператору, расположенному за переключателем *switch*.

Если требуется организовать использование какого-либо логического блока операторов при выполнении *нескольких* условий, то в этом случае впереди указанного блока операторов все метки *case* вместе с их константами перечисляют, разделяя двоеточием. Приведём иллюстративный пример:

```
switch (selektor)
{
    case Znachenie_1 :
    case Znachenie_2 :
        {Блок операторов 1}
        break;
    case Znachenie_3 :
    case Znachenie_4 :
        {Блок операторов 2}
        break;
    default: {Блок операторов 3}
} //switch
```

Число меток *case* в операторе *switch* не ограничивается, они могут вообще отсутствовать. Но вот метка *default* может либо отсутствовать, либо присутствовать в *единственном* экземпляре.

3.6. Дополнительные возможности командной кнопки

Практически во всех приложениях *Windows* большинство команд можно ввести как при помощи меню и пиктограмм, так и с использованием горячих клавиш (клавиш ускоренного или быстрого доступа). При этом необходимая горячая клавиша, нажатие на которую совместно с клавишей *Alt* заменяет ввод соответствующей команды, обычно подчёркнута в названии этой команды на интерфейсе. Таким образом, интерфейс приложения с пунктами меню выступает в роли справочной системы, которая всегда под рукой, лучше сказать: *перед глазами*. При наборе данных в полях ввода используется только клавиатура, поэтому и команды вводить с её использованием значительно удобнее: пользователю не

нужно отрывать руки от клавиш с тем, чтобы ввести команду при помощи мыши. Профессионалы в большинстве приложений работают в основном с использованием клавиатуры.

По этой причине при разработке приложений следует расставлять упомянутые подсказки на интерфейсе. Буква в заголовке объекта становится подчёркнутой и получает свойства горячей клавиши, если впереди неё в поле ввода свойства *Caption* поместить знак амперсанта &. Размещение & перед заголовком *Пуск* командной кнопки (типа *TButton* или *TBitBtn*) приведёт к тому, что эта кнопка

будет выглядеть вот так:



Теперь ввод команды *Alt+П* эквивалентен нажатию командной кнопки *Пуск* мышью. Однако нажимать прописную букву *П* не всегда удобно, ведь в этом случае требуется переводить клавиатуру в верхний регистр. Поэтому в слове *Пуск*

можно подчеркнуть, например, вторую литеру:



Имеется возможность *нажимать* командную кнопку также клавишей *Enter*. Для реализации такого удобства в свойстве *Default* объекта типа *TButton* или *TBitBtn* значение *false*, установленное по умолчанию, необходимо изменить на *true*. Теперь нажатие клавиши *Enter* будет эквивалентно нажатию командной кнопки, даже в случае, когда эта кнопка не находится в фокусе ввода. Если имеется несколько командных кнопок, то клавиша *Enter* активизирует ту из них, которая находится в фокусе ввода. Напоминаем, что фокус ввода перемещается по объектам управления интерфейса при помощи клавиши *Tab* (или мыши).

Вопросы для самоконтроля

- ☐ Укажите отличия подключения пользовательских и стандартных заголовочных файлов к файлу с реализациями функций проекта.
- ☐ Чем опасна операция присваивания значения вещественной переменной или константы целочисленной переменной?
- ☐ Какой тип имеет результат математического выражения, в котором участвуют переменные и константы разных числовых типов?
- ☐ Почему недопустим следующий фрагмент кода программы?

```
int x= 1;
float y= 2, z= 3.52;
x= z%y;
```
- ☐ Что имеет больший приоритет сложение и вычитание или умножение и деление?
- ☐ Как следует поступить в том случае, когда вы не помните приоритет отдельных операций?
- ☐ Укажите причину, из-за которой в операции сравнения на равенство нельзя использовать вещественные переменные и константы.
- ☐ Каким образом в операторе множественного выбора *switch* осуществить выполнение одного и того же логического блока при нескольких константах выбора?

3.7. Задача для программирования

Высота H подъема тела, брошенного под углом α к горизонту с начальной скоростью V , определяется выражением

$$H = V^2 \cdot \sin^2(\alpha) / 2g,$$

где g – ускорение свободного падения тела. Требуется разработать программу расчёта H по произвольным значениям V и α . Внешний вид приложения с полями ввода исходных данных, полем для вывода результата расчёта и командной кнопкой аналогичен интерфейсу, показанному на рис. 3.1.

3.8. Вариант решения задачи

Здесь приведена только функция, реализующая расчёт (после нажатия клавиши *Расчёт*) по введенным значениям V и α высоты подъёма тела H .


```
void __fastcall TDalnostj_Poleta::RaschetClick(TObject *Sender)
{
    const float g= 9.81, Pi= 3.141593;
    float fV, fAlfa, fAlfaRad, fH;
    //fV Начальная скорость тела в м/с
    //fAlfa - угол к горизонту вектора начальной
    //скорости тела в градусах
    //fAlfaRad -значение fAlfa в радианах
    //fH высота подъёма тела в м
    fV= StrToFloat(V->Text);
    fAlfa= StrToFloat(Alfa->Text);
    if (fV>= 0 && fAlfa>= 0 && fAlfa<= 180)
    {
        fAlfaRad= fAlfa*Pi/180.0;
        fH= pow((fV*sin(fAlfaRad)), 2.0)/(2*g);
        H->Caption= FloatToStrF(fH, ffFixed, 10, 2);
    }//if
    else
    {
        H->Caption= "";
        ShowMessage("Введены недопустимые значения");
    }//else
}//RaschetClick
```

Урок 4. Все о циклах

4.1. Зачем нужны циклы?

В научных и инженерных исследованиях часто возникает необходимость в расчётах функциональных зависимостей в заданных *диапазонах* значений их аргументов. На основе полученных знаний вы уже способны проводить упомянутые расчёты. В подтверждение этого утверждения приведём программу *Povtor*, решающую простую задачу: требуется вычислить значения функции $y = a \cdot x^2$ в диапазоне изменения аргумента x от 0 до 10 с шагом $dx = 2$ для значения константы $a = 2$.

В приложении, которое реализует решение этой задачи, используем визуальный компонент интерфейса *StringGrid* (таблицу строк), он находится на вкладке

Additional (дополнительные) *Палитры Компонентов* и выглядит вот так: . Этот объект позволяет осуществлять ввод и вывод *текстовых* значений в клетки или ячейки (*Cells*) таблицы. В одной колонке ячеек этой таблицы (назовём её x) поместим значения аргумента, а в другой (с именем y) расположим соответствующие значения вычисленной функции.

Выберете этот компонент на *Панели Компонентов*, сделайте из него объект интерфейса. В результате вы увидите таблицу с полосами прокрутки по горизонтали и вертикали (см. рис. 4.1). Заголовок формы *Повтор*, а её имя – *Povtor_Vuch* (повтор вычислений).

Далее сделайте активным этот единственный компонент формы (щёлкните по нему) и перейдите в *Инспектор Объектов*, где на вкладке *Свойства* найдите свойство *ColCount* (число столбцов). Все пять колонок, установленные по умол-

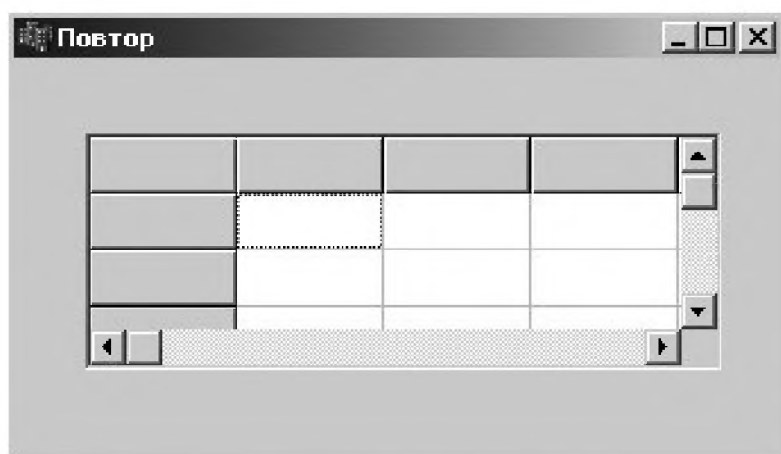


Рис. 4.1. Вид объекта *StringGrid*

чанию, в рассматриваемой задаче не нужны, для размещения значений аргумента и функции достаточно два столбца.

Нумерация строк и столбцов в объектах типа *TStringGrid* начинается с нуля, поэтому оставим только нулевой и первый столбцы. Нулевой столбец и нулевая строка (на рис 4.1 они выделены серым цветом) обычно используются для вывода заголовков соответственно строк и столбцов (например, для указания их номера). При помощи полос скроллинга можно прокручивать строки и столбцы (если все они не помещаются в окне таблицы строк), однако при такой прокрутке заголовочные столбцы и строки остаются *неподвижными*, что очень удобно при работе с большой таблицей данных.

В разрабатываемой программе заголовки строк использовать не будем, поэтому отключим вывод заголовочного столбца. Для этого в свойстве *FixedCols* (фиксированные столбцы) установим нулевое значение. Если в поле ввода этого свойства ввести 1 или 2, то скроллинг будет осуществляться соответственно со столбца с номером 1 или 2 (фактически это будут второй и третий столбцы). В свойстве *ColCount* вместо числа 5 поставьте требуемое в задаче число 2. Можно установить любой цвет заголовочных столбцов и строк при помощи свойства *FixedColor* (установка цвета).

Число значений аргумента (и функции) в нашей задаче равно шести: 0, 2, 4, 6, 8 и 10. Поэтому с учётом нулевой строки для заголовков столбцов в свойстве *RowCount* (число строк) вместо числа 5 поставьте 7. В свойстве *Name* вместо *StringGrid1* наберите имя *Rezytlat*, ведь в таблице будут представлены *результаты* вычислений.

В свойстве *Cursor* при помощи раскрывающегося списка выберете форму курсора в виде руки (*crHandPoint*). Зачем? И почему именно в виде руки? В целях обучения потренируйтесь выбирать свойства объектов. Впрочем, можете не «играть» с этим свойством. Методом протаскивания увеличьте высоту и уменьшите ширину объекта до размеров, необходимых для полного вывода всех ячеек. После этого исчезнут полосы прокрутки.

Далее поместите на форме командную кнопку с заголовком *Пуск* и именем *Pysk*. Наконец, сделайте двойной щелчок по этой кнопке и начните программировать задачу. Ниже показана функция *PyskClick*, в которой введены только константы задачи *a*, *dx* и *x_Min* да целочисленные переменные *ij*, *ix*, *iy*. Первая буква в этих переменных подсказывает их тип *int*. Переменная *ij* предназначена для хранения номера строки.

```
void __fastcall TPovtor_Vuch::PyskClick(TObject *Sender)
{
    const int a= 2, dx= 2, x_Min= 0;
    int ix, iy, ij;// ij- номер строки
} //PyskClick
```

У объектов типа *TStringGrid* имеется свойство *Cells* для доступа к выбранной ячейке таблицы. Такой доступ осуществляется указанием в квадратных скобках номера столбца и номера строки: *Cells[Nomer_Stolbca][Nomer_Stroki]*. Для вывода заголовков столбцов на нулевой строке таблицы в программе добавьте строки:

```
Rezyltat->Cells[0][0]= "X";
Rezyltat->Cells[1][0]= "Y";
```

Доступ к свойству *Cells* выполняется при помощи указателей (операция стрелка). Как отмечалось выше, в таблице строк данные хранятся в *строковом* виде, в виде переменных типа *String*. Поэтому при копировании данных из ячеек объектов типа *TStringGrid* в переменные *числовых* типов необходимо преобразование данных, например с использованием функций *StrToInt()* или *StrToFloat()*. Однако *копирование* числовых (и символьных) значений или переменных в ячейки таблицы строк можно осуществлять *без их преобразования* к типу *String*. Это оказывается возможным, поскольку тип *String* на самом деле является классом, конструктор которого осуществляет необходимые преобразования указанных типов данных к строковому формату. Что собой представляет конструктор класса, рассказывается на 14 уроке. Сейчас же запомните, что, например, две следующие строки кода приводят к *одинаковым* результатам.

```
Rezyltat->Cells[0][ij]= IntToStr(ix);
Rezyltat->Cells[0][ij]= ix;
```

Ниже будем применять более компактный вариант. Для расчёта и размещения в таблице первого значения аргумента (0) и значения функции (0) при этом аргументе можно записать следующие четыре строки операторов:

```
ij= 1; ix= x_Min + (ij-1)*dx; //ix= 0
Rezyltat->Cells[0][ij]= ix;
iy= a*pow(ix, 2); //iy= 0
Rezyltat->Cells[1][ij]= iy;
```

В начале файла *Povtor.cpp*, содержащего функцию *PyskClick* (для правомерности использования функции *pow*), укажите директиву препроцессорной обработки:

```
#include <math.h>
```

Трассировкой приведенного кода удостоверьтесь в том, что в *ix* и *iy* находятся нули. Следующий шаг вычислений осуществляется аналогичными строками:

```
ij=ij + 1; ix=x_Min + (ij-1)*dx; //ij= 2, ix= 0 + 2= 2
Rezyltat->Cells[0][ij]= ix;
iy= a*pow(ix,2); //iy= 8
Rezyltat->Cells[1][ij]= iy;
```

Полный код этой функции показан ниже.

```
void __fastcall TPovtor_Vuch::PyskClick(TObject *Sender)
{
    const int a= 2, dx= 2, x_Min= 0; //x_Min - минимальное значение x

    int ix, iy, ij; //ij- номер строки
    //Называем столбцы
    Rezyltat->Cells[0][0]= "X";
    Rezyltat->Cells[1][0]= "Y";

    //Вычисляем ix и iy
    ij=1; ix= x_Min + (ij - 1)*dx; //ix= 0
    Rezyltat->Cells[0][ij]= ix;
    iy= a*pow(ix, 2); //iy= 0
```



```

Rezyltat->Cells[1][ij]= iy;

ij=ij + 1; ix=x_Min + (ij-1)*dx;//ij= 2, ix= 2
Rezyltat->Cells[0][ij]= ix;
iy= a*pow(ix, 2); //iy= 8
Rezyltat->Cells[1][ij]= iy;

ij=ij + 1; ix=x_Min + (ij-1)*dx;//ij= 3, ix= 4
Rezyltat->Cells[0][ij]= ix;
iy= a*pow(ix, 2); //iy= 32
Rezyltat->Cells[1][ij]= iy;

ij=ij + 1; ix=x_Min + (ij-1)*dx;//ij= 4, ix= 6
Rezyltat->Cells[0][ij]= ix;
iy= a*pow(ix, 2); //iy= 72
Rezyltat->Cells[1][ij]= iy;

ij=ij + 1; ix=x_Min + (ij-1)*dx;//ij= 5, ix= 8
Rezyltat->Cells[0][ij]= ix;
iy= a*pow(ix, 2); //iy= 128
Rezyltat->Cells[1][ij]= iy;

ij=ij + 1; ix=x_Min + (ij-1)*dx; // ij= 6, ix= 10
Rezyltat->Cells[0][ij]= ix;
iy= a*pow(ix, 2); //iy= 200
Rezyltat->Cells[1][ij]= iy;
} //PyskClick

```

Результат работы приложения показан на рис. 4.2.

Как очевидно из условия задачи, все значения переменных *ix* и *iy* являются целыми числами, поэтому для их описания использован целочисленный тип *int*. Зачем придуман целочисленный тип данных? Разве нельзя обойтись вещественными числами, ведь целые числа полностью входят в него? Дадим ответы на эти вполне резонные вопросы. Для хранения чисел типа *int* необходима ячейка памяти по объёму в *два* раза меньшая, чем для вещественного типа *double*: для типа *double* требуется 8 байт, а для *int* – лишь 4 байта. Кроме того, компьютер все операции с целыми числами выполняет приблизительно в 100 раз *быстрее*. Поэтому имеется преимущество применения переменных типа *int* не только в сравнении с типом *double*, но и *float*, у которого размер ячейки такой же, как и у типа *int*.

Читатель может возразить: Разве для нашей простенькой задачи это так уж важно? Да, для этой *элементарной* задачи экономия памяти на $4 \cdot 2 = 8$ байт и уменьшение времени работы программы никакого *принципиального* значения не имеют – на *любом* компьютере эта программа выполняется практически мгновенно. Но мы умышленно рассмотрели очень простую задачу для наглядности описания конструкций языка. В более сложных задачах такая, на

The screenshot shows a Windows 95 desktop environment. At the top, there is a taskbar with a 'Повтор' (Repeat) window. The window contains a table with two columns, X and Y, and six rows of data. Below the table is a button labeled 'Пуск' (Start).

X	Y
0	0
2	8
4	32
6	72
8	128
10	200

Пуск

Рис. 4.2. Результат работы приложения

первый взгляд, мелочная экономия часто приводит к существенным преимуществам. Поэтому *настоятельно рекомендуем* при разработке программ экономить оперативную память и повышать быстродействие программ. Приучайтесь делать это *автоматически*, как моете руки перед едой.

Расчёт и вывод на экран всех значений *iy* и *ix* осуществляется повтором одних и тех же операторов. Поэтому программа названа *Povtor*. При увеличении интервала *x*, например в 100 раз, во столько же раз увеличится и тело программы. Поэтому такой метод решения задачи очень неразумен. Для выполнения повторяющихся действий во всех языках программирования применяются операторы циклов или повторов. Это предложение является ответом на вопрос, сформулированный в названии урока. Полигоном для изучения циклов в *C++Builder* послужит задача, сформулированная в начале урока.

Но прежде, чем на полигоне начнутся учения различных родов войск – видов циклов, приведём важную рекомендацию, она является *стандартом* современных структурированных программ: никогда не помещайте конкретные цифровые значения в тело программы. Программы должны допускать возможность быстрого изменения *всех* своих параметров: коэффициентов, шагов дискретизации, начальных, конечных значений аргументов и т. д. Программа, составленная с использованием символьных обозначений для своих коэффициентов, легко читается, очень просто модернизируется – поэтому оказывается пригодной для значительно более широкого использования. При этом также существенно сокращается время на её разработку и отладку. Иногда говорят, что такая программа легко *масштабируется*.

Однако возможность быстрой замены констант в нашем проекте имеется только у программиста. Пользователь же программы, располагающий только её исполняемым файлом, вынужден довольствоваться одними и теми же значениями, навсегда заданными в разделе описания констант. Поэтому такая программа не будет пользоваться большим спросом.

Для ввода *произвольных* значений констант *a* и *dx* используем объект *Поле ввода* (компонент *Edit*) совместно с объектом *Кнопка Больше-Меньше* (*UpDown*)



Далее, на вкладке *Win32 Палитры Компонентов* активизируйте компонент *UpDown* и щелкните по форме. После этого в свойстве *Increment* (увеличение) появившегося на форме объекта *UpDown1* оставьте генерируемое по умолчанию значение 1 (инкремент, шаг приращения, может быть *только* целым числом). В свойствах *Max* и *Min* объекта *UpDown1* значения по умолчанию соответственно равны 100 и –100. Они определяют минимальную и максимальную границы из-

менения величины, вводимой в поле ввода, которое обслуживает эта кнопка. Для свойства *Associate* (объединить) выберете поле ввода *Znachen_a*. В результате кнопка *UpDown1* присоединится к правой границе (по умолчанию) этого объекта, габариты кнопки окажутся согласованными с габаритами поля ввода. Теперь кнопка может управлять полем ввода: после запуска программы щелчками по стрелкам этой кнопки увеличиваются или уменьшаются с дискретностью в единицу в интервале от -100 до 100 значение константы a от стартового значения 2.

Свойству *AlignButton* (выравнивание кнопки) можно установить одно из значений *unLeft* или *unRight*, (*un* – от *union* – объединение), что позволяет расположить кнопку слева или справа от поля ввода. Свойство *Orientation* может принимать значения *udVertical* или *udHorizontal*. При этом стрелки кнопки располагаются по вертикали (одна под другой) или по горизонтали (одна рядом с другой).

Свойство *ArrowKeys* определяет, будет ли управлять кнопкой клавиши клавиатуры со стрелками. Для лучшего восприятия чисел (абсолютные значения которых превышают 999) можно при помощи свойства *Thousands* (тысячи) между тремя цифрами разрядов вводимого числа автоматически вставлять разделительные пробелы. Свойство *Position* позволяет установить *исходное* значение числа (его можно записать и в свойстве *Text* обслуживаемого кнопкой поля ввода).

Если в свойстве *Wrap* (граница) установит *false*, то при достижении нижней или верхней границы заданного диапазона вводимое число фиксируется на предельном значении и дальнейшее нажатие кнопки ни к чему не приведёт. Если же в этом свойстве установить *true*, то попытка превысить максимум приведёт к сбросу вводимого значения к минимуму, желание уменьшить минимум – сбросит его к максимуму. Таким образом, изменение чисел закольцовано.

Свойство *ReadOnly* является совместным свойством объединённых объектов типа *TEdit* и *TUpDown*. По умолчанию в нём установлено значение *false*, что позволяет в поле ввода вводить любое, в том числе и нечисловое значение. При константе выбора *true* *числовые* значение поля ввода можно изменять *только* клавишами с соответствующими стрелками, исходные нечисловые значения поля ввода при запуске приложения заменяются нулевым значением.

Далее породите кнопку *Больше-Меньше UpDown2* и объедините её с поля ввода *Znachen_dx*. Над упомянутыми полями ввода выведете заголовки: *Ввод a* и *Ввод dx*. В итоге интерфейс приложения может выглядеть, например, так, как показано на рис. 4.3. Над таблицей расположено окно вывода с расчётной формулой.

Раздел описания констант программы, с учётом преобразования введенных текстовых

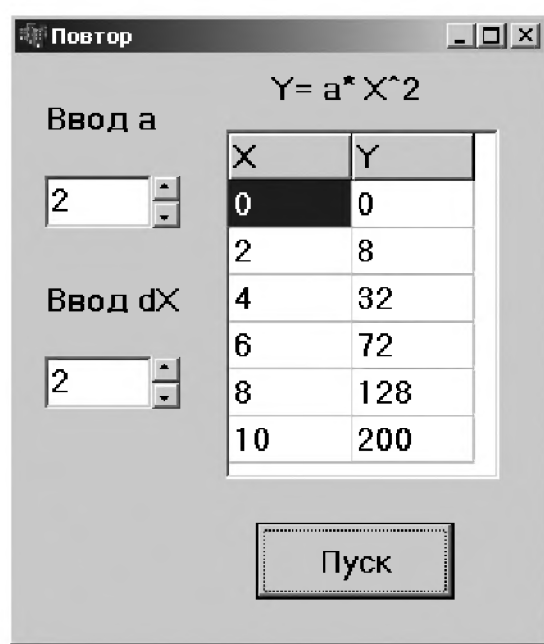


Рис. 4.3. Интерфейс приложения *Повтор* после модернизации

значений к целочисленным значениям соответствующих констант выглядит теперь так:

```
const int a= StrToInt(Znachen_a->Text),
          dx= StrToInt(Znachen_dx->Text),
          x_Min= 0; //Минимальное значение ix;
```

Теперь перейдём к основной цели нашего урока – к циклам.

4.2. Оператор цикла *for*

В задачах, в которых заранее известно (или легко определить) число повторений какого-либо процесса, обычно применяется оператор цикла (или повтора) *for* (для). В нашей задаче величину *y* требуется вычислить 6 раз, для *x*: 0, 2, 4, 6, 8, 10. Поэтому для её решения целесообразно применить цикл *for*, а программу назвать *O_for* (оператор *for*). Сделайте новую директорию с именем *Cikl_for*, скопируйте в неё все файлы предшествующего проекта, переименуйте заголовок формы и её имя на *O_for*, имя файла *Povtor.cpp* замените именем *Oper_for.cpp* (для этого используйте команду горизонтального меню *File/Save As*), а имя файла *ProjectPovtor.cpp* – на *ProjectO_for.cpp* (при помощи команды *File/Save Project As*). Тело функции *PyskClick* теперь смените вариантом, который показан ниже.

```
void __fastcall TO_for::PyskClick(TObject *Sender)
{
    const int a= StrToInt(Znachen_a->Text),
              dx= StrToInt(Znachen_dx->Text),
              x_Min= 0,
              n= 6;

    int ix, iy, ij; // ij- номер строки
    //Называем столбцы
    Rezyltat->Cells[0][0]= "X";
    Rezyltat->Cells[1][0]= "Y";

    //Вычисляем ix и iy
    for (ij= 1; ij<= n; ij= ij + 1)
    {
        ix= x_Min + (ij - 1)*dx;
        Rezyltat->Cells[0][ij]= ix;
        iy= a*pow(ix, 2);
        Rezyltat->Cells[1][ij]= iy;
    } //for
} //PyskClick
```

В этом проекте интерфейс программы не изменился, в коде программы введена дополнительная константа $n = 6$, а длинная последовательность операторов, обеспечивающих расчёт и показ всех значений *x* и *y*, заменена компактным циклом *for*. Давайте рассмотрим его устройство.

После служебного слова *for* в круглых скобках находятся операторы, *управляющие* работой цикла, а ниже, в фигурных скобках, расположено *тело цикла*: совокупность операторов, выполнение которых повторяется на *каждом* шаге цикла. Ключевое слово *for* вместе с содержимым круглых скобок называется *заголовком*

В теле цикла находятся *все* операторы, которые последовательно повторялись в функции *PyskClick* из предшествующего проекта. Работа же по изменению переменной *ij* производится в заголовке цикла *for*. Управляющие операторы цикла (напоминаем, они находятся в круглых скобках) разбиты на три части (или раздела), отделяемые друг от друга точкой с запятой.

```
for (int ij = 1; ij<= n; ij= ij + 1)
```

Во втором разделе находится оператор (или операторы), определяющие условие продолжения цикла ($ij \leq n$), оно проверяется на *каждом* шаге цикла. Если это условие не выполняется уже перед началом цикла, то оператор *for* пропускается (игнорируется). В нашей задаче $n = 6$, на первом шаге $ij = 1$, поэтому условие $1 \leq 6$ выполняется и первый шаг цикла осуществится. В результате рассчитываются и выводятся на экран значения $x = 0$ и $y = 0$ (в первой строке таблицы будут выведены нули). Второй раздел называется *разделом проверки продолжения цикла*.

Таким же образом осуществится расчёт на третьем, четвёртом, пятом и шестом шагах цикла. На шестом шаге ($ij=6$) выводятся значения: $x=10$ и $y=200$. После его завершения значение ij увеличится на единицу и станет равным 7. В этом случае условие $ij \leq n$ ($7 \leq 6$) нарушается, поэтому цикл завершает свою работу.

```
const int x_Max= 10;//Максимальное значение аргумента
for (int ij= 1, ix= x_Min, iy; ix<= x_Max; ij= ij+1, ix= ix+dx)
{
```

```

Rezyltat->Cells[0][ij]= ix;
iy= a*pow(ix, 2);
Rezyltat->Cells[1][ij]= iy;
} //for

```

В этом примере в третьем разделе заголовка цикла находится два оператора, они разделяются *запятыми*. В разделе инициализации производится объявление переменных *ij*, *ix* и *iy*, переменные *ij* и *ix* получают начальные (стартовые) значения. Все переменные, объявленные в разделе инициализации, видимы только в заголовке цикла и в его теле.

В окончательном варианте программы на интерфейсе следует добавить поля для ввода минимальных и максимальных значений аргумента и автоматизировать подсчёт числа шагов цикла *n*. Интерфейс приложения для этого случая показан на рис. 4.4, а функция *PyskClick* имеет вид:



Рис. 4.4. Окончательный интерфейс приложения *O_for*

```

void __fastcall TO_for::PyskClick(TObject *Sender)
{
    const int a= StrToInt(Znachen_a->Text),
              dx= StrToInt(Znachen_dx->Text),
              x_Min= StrToInt(Znachen_X_Min->Text),
              x_Max= StrToInt(Znachen_X_Max->Text),
              n= (x_Max - x_Min)/dx + 1;
    int ix, iy, ij; // ij- номер строки
    //Называем столбцы
    Rezyltat->Cells[0][0]= "X";
    Rezyltat->Cells[1][0]= "Y";
    //Вычисляем ix и iy
    for (ij= 1; ij<= n; ++ij)
    {
        ix= x_Min+ (ij-1)*dx;
        Rezyltat->Cells[0][ij]= ix;
        iy= a*pow(ix, 2);
        Rezyltat->Cells[1][ij]= iy;
    } //for
} //PyskClick

```

Операции по модернизации интерфейса и раздела описания констант уже не раз выполнялись ранее. Поэтому далее подробно рассмотрим только раздел изменения управляющих операторов цикла и формулу, при помощи которой вычисляется *n*.

В результате арифметических операций над целочисленными величинами $(x_Max - x_Min)/dx$ получится целое число (см. п. 3.3.4), в ходе получения этого числа дробная часть истинного частного будет отброшена. В случае, например, когда $dx=3$ истинное отношение составит $(10 - 0)/3 = 3,3333\dots$, а программой будет вычислено значение 3. Но при таком шаге число шагов равно не трём, а четы-

рём (0, 3, 6, 9). Поэтому, добавив в операторе присваивания единицу, мы получим правильное число шагов. В нашем примере, при $dx = 2$, отношение точно равно 5, добавляемая единица снова вносит корректировку: мы получаем требуемое число 6. Реализованная в программе формула $n = (x_Max - x_Min)/dx + 1$ всегда верно определяет число шагов цикла.

В третьем разделе заголовка цикла находится оператор $++ij$, он предназначен для изменения переменной ij . В п. 3.3.4 указывалось о том, что эта операция называется операцией инкремента. Унарные операции инкремента ($++$) или декремента ($--$) сводятся соответственно к увеличению или уменьшению значения операнда на единицу. Эти операции могут размещаться как перед переменной (префиксная форма записи), так и позади неё (постфиксная форма записи).

При префиксной форме значение переменной вначале увеличивается или уменьшается на единицу, а затем её новое значение используется в выражении. Поэтому операторы присваивания $ij = ij + 1$; и $ij = ++ij$; эквивалентны, операторы $ij = ij - 1$; и $ij = --ij$; также полностью равносильны.

При постфиксной форме в выражении используется текущее значение переменной, и только после завершения вычисления её значение увеличивается или уменьшается на единицу: $ij = ij++$; $ij = ij--$;

В заголовке цикла четыре варианта $ij = ij + 1$, $ij = ++ij$, $++ij$ и $ij++$ приведут к одним и тем же результатам. Почему же так происходит в двух последних случаях? Всё дело в том, что в последних двух вариантах *нет* никакого *выражения*. Процедурные же варианты использования обеих форм $++ij$ и $ij++$ увеличивают значение ij на единицу после *каждого* шага цикла.

Давайте теперь для закрепления сказанного поэкспериментируем. Вначале заголовок цикла изменим так:

```
for (ij= 1; ij<= n; ij= ++ij)
```

Проверка покажет, что приложение функционирует также как и ранее: после каждого шага цикла значение ij увеличивается на единицу, что позволяет осуществить расчёт последующей строки таблицы $x-y$.

Но, что произойдёт при такой модернизации заголовка цикла:

```
for (ij= 1; ij<= n; ij= ij++)?
```

Читатели, которые не только трассировкой в уме испытывают все предлагаемые изменения кода программ, поступали абсолютно правильно: полное глубоко осознанное понимание кода приходит лишь после *практического* его *испытания* (запуска) на компьютере. Критерием истины может быть только практика. Однако вариант, который предложен выше, не спешите запускать на ПК. Давайте вначале подвергнем его *ручной* трассировке, ей всегда надо отдавать предпочтение перед бездумным эмпиризмом судорожного нажатия клавиш.

После выполнения первого шага цикла, при $ij = 1$, подойдёт очередь исполнения оператора $ij = ij++$. Как уже отмечалось выше, в случае постфиксной формы инкремента вначале будут выполнены *все* операции, в которых участвует переменная ij , а затем только произойдёт увеличение значения ij на единицу. Однако в

нашем примере очередь выполнения этой *второй* части операции *никогда* не подходит.

Судите сами. В результате первой части постфиксной операции *повторно* выполнится тело цикла, и эстафетная палочка его операций переместится во второй раздел заголовка, где осуществится проверка условия выполнения следующего шага цикла. Напоминаем что первый, инициализирующий, раздел выполняется только один раз, при входе в цикл. Проверка условия $1 < 6$ окажется успешной (*true*), поэтому произойдёт выполнение цикла при $ij = 1$ *третий* раз, после опять очередь подойдёт к оператору $ij = ij++$. И так далее до бесконечности. В таких случаях говорят, что программа *зациклилась* или *зависла*. Если вы решили всё же эти заключения проверить экспериментально, то для прекращения работы программы перейдите в редактор кода и введите команду *Ctrl+F2 (Run/Program Reset)*, либо перезапустите компьютер.

Почему же так происходит? По ходу выполнения цикла постфиксная форма инкремента «забывает» совершить свой второй шаг. Этот нюанс *Builder* знать весьма полезно. Для корректной *практической* проверки этих рассуждений запустите функцию в таком варианте.

```
void __fastcall TO_for::PyskClick(TObject *Sender)
{
    const int n= 6;
    int is= 0;
    for (int ij= 1; ij<= n; ij= ij++)
    {
        if (is> 100)
        {
            ShowMessage(is);
            break;
        }
        is= is + 1; //или ++is; или is++;
    }
} //PyskClick
```

Здесь введена и инициализирована нулевым значением целочисленная переменная *is*. Оператор *if* после выполнения условия $is > 100$ осуществляет при помощи оператора *break* (см. подраздел 3.5) выход из цикла. В ходе первого шага цикла исходное нулевое значение переменной *is* увеличится на единицу и становится равным единице ($1 = 0 + 1$). После второго шага в *is* окажется 2, после третьего – 3. Работа операторов $is = is + 1$ и $ij = ij + 1$ полностью аналогична. Оператор $is = is + 1$ также можно заменить переменной с инкрементами $++is$ или $is++$. Такие операторы называются *счётчиками*, они *очень* широко применяются при разработке программ. В переменной *is* подсчитывается число шагов цикла *for*.

После запуска программы увидите сообщение «101». Функция *PyskClick* 101 раз запустила оператор $is = is + 1$ (при этом $ij == 1$) и только после этого работа цикла (и функции) остановлена содружеством операторов *if* и *break*.

Вместо оператора *break* для этих же целей можно использовать, например, оператор $ij = n + 1$, в результате работы которого перед последующим шагом цикла окажется, что $ij == n + 1$ и цикл будет завершён. Применение функции *Abort()* вме-

Если же вы не желаете выходить из цикла, а только прервать выполнение его *конкретного* шага, на котором оказалось выполненным какое-то заранее заданное условие, то для таких ситуаций следует использовать оператор *continue* (продолжать), его можно использовать *только* в операторах цикла. С помощью этого оператора принудительно завершается *текущий* шаг цикла. Далее осуществляется проверка условия выполнения последующего шага цикла точно так же, как и после стандартного завершения обычного шага. Для иллюстрации работы оператора *continue* предположим, что в разрабатываемой программе *не требуется* выводить значения *iy*, находящиеся в интервале: $32 \leq y \leq 72$. В этом случае код оператора *for* функции *PyskClick* изменится так:

В результате в колонке таблицы с именем *y* будут исключены два значения 32 и 72.

В начале настоящего раздела упоминалось о том, что цикл *for* предназначен в первую очередь для использования в тех задачах, в которых можно заранее установить число шагов (или итераций) цикла. Однако, с использованием оператора *break* или функции *Abort()* цикл *for* вполне можете применить и в тех ситуациях, когда необходимое число шагов цикла определяется *только* по ходу решения задачи, после выполнения заданного условия.

Операторы в разделах заголовка цикла могут отсутствовать, однако их место необходимо выделять точкой с запятой, роль отсутствующих операторов должны выполнять операторы, расположенные в других местах программы. Вот так, например, будет выглядеть цикл *for* функции *PyskClick* в случае отсутствия оператора в разделе инициализации и разделе изменения управляющей переменной:

Как видите, для организации работы цикла инициализация переменной *ij* осуществляется *перед* циклом, а её инкремент выполняется в конце тела цикла.

Вот такой цикл называется вечным, при его реализации программа заикнется:

```
for ( ; ; )
{
} //for
```

Это происходит потому, что каждый пустой раздел заголовка эквивалентен значению *true*, а тело цикла является пустым (нет управляющего параметра). Такого типа циклы иногда используют для осуществления вычислений, завершить которые необходимо после выполнения заданного условия. В этом случае инициализация параметра цикла осуществляется перед циклом, оператор, контролирующий продолжение цикла, следует расположить в теле оператора *for*. Приведём поясняющий пример:

```
int iS= 0;
for ( ; ; )
{
    if iS>100
        break;
    iS++;
} //for
ShowMessage(iS); //iS== 101
```

В заключение этого раздела вернёмся к исходной задаче и обобщим её на случай вещественных значений *всех* констант. Модернизированная функция *PyskClick* теперь выглядит так:

```
void __fastcall TO_for::PyskClick(TObject *Sender)
{
    const float a= StrToFloat(Znachen_a->Text),
                dx= StrToFloat(Znachen_dx->Text),
                x_Min= StrToFloat(Znachen_X_Min->Text),
                x_Max= StrToFloat(Znachen_X_Max->Text);
    const int n= (x_Max - x_Min)/dx + 2;
    float fx, fy;
    int ij; //ij- номер строки
    //Называем столбцы
    Rezyltat->Cells[0][0]= "X";
    Rezyltat->Cells[1][0]= "Y";
    //Вычисляем fx и fy
    for (ij= 1; ij<= n; ++ij)
    {
        fx= x_Min+ (ij - 1)*dx;
        Rezyltat->Cells[0][ij]= FloatToStrF(fx, ffFixed, 10, 2);
        fy= a*pow(fx, 2);
        Rezyltat->Cells[1][ij]= FloatToStrF(fy, ffFixed, 10, 2);
    } //for
} //PyskClick
```

Интерфейс приложения не изменит своего вида, однако мы произведём настройку полей его ввода таким образом, чтобы в них можно было заносить *произвольные* вещественные значения, а не только те, которые допускаются указанным диапазоном *Min* – *Max* и инкрементом кнопки *Больше-меньше*. С этой целью во всех экземплярах объекта типа *TEdit*, связанных с объектом типа *TUpDown*, для свойства *ReadOnly* установим значение *false*. Пользователь, к сожалению, в этом

Вопросы для текущего самоконтроля

- ### 4.3. Оператор цикла *while*

Этот оператор цикла пригоден и в задачах с заранее известным числом итераций. Поэтому применим его для прежней учебной задачи. Ниже приведен код функции *PuskClick* с циклом *while*. Переименуйте приложение *O_for* в *O_while* и сохраните его в директории *Cikl_while*. В новом проекте измените код функции *PuskClick*.

```
void __fastcall TO_while::PushClick(TObject *Sender)
{
    const int a= StrToInt(Znachen_a->Text),
              dx= StrToInt(Znachen_dx->Text),
              x_Min= StrToInt(Znachen_X_Min->Text),
              x_Max= StrToInt(Znachen_X_Max->Text);
    int ix, iy, ij;// ij - номер строки
    //Называем столбцы
    Rezyltat->Cells[0][0]= "X";
    Rezyltat->Cells[1][0]= "Y";
    //Вычисляем ix и iy
    ij= 1; ix= x_Min;//Начальные значения для входа в цикл
    while (ix<= x_Max)//Предусловие
    {
        Rezyltat->Cells[0][ij]= ix;
        iy= a*pow(ix, 2);
        Rezyltat->Cells[1][ij]= iy;
        ++ij;//Переход на новую строку таблицы
        ix= ix + dx;
    }
}
```

```
    }//while  
} //PuskClick
```

Цикл приступает к работе *лишь в случае*, тогда выполняется условие входа в цикл: $ix \leq x_Max$ (условие должно быть заключено в круглые скобки). Поскольку в нашем примере перед циклом аргументу ix было присвоено начальное значение x_Min , которое удовлетворяет условию входа в цикл, то цикл будет запущен на выполнение и осуществится его первый шаг. На первом шаге, так же, как и в цикле *for*, для $ix = x_Min$ вычисляется значение iy , и выводятся результаты: $x = 0$ $y = 0$.

В конце тела цикла оператор $++ij$ увеличит на единицу номер обрабатываемой строки таблицы, оператор $ix = ix + dx$ увеличит аргумент ix на величину приращения dx . В результате первый шаг цикла будет завершён при $ij = 1 + 1 = 2$ и $ix = x_Min + dx = 0 + 2 = 2$.

Так уж устроен цикл *while*, что по завершению каждого своего шага он стремится к продолжению своей работы, в этом он настоящий трудоголик. Однако он – весьма прагматичен, так как продолжает трудиться лишь тогда, когда трудовой договор не нарушается – выполняется условие входа в цикл. Поскольку перед началом второго шага этого цикла условие $ix \leq x_Max$ ($2 < 10$) выполняется, поэтому цикл *while* честно отработает ещё одну свою смену: распечатаются новые значения ix и iy , а переменные ij и ix увеличатся на величины своих приращений. В результате окажется, что $ij = 2 + 1 = 3$ и $ix = 2 + 2 = 4$. Новое значение аргумента опять устраивает привратника на входе в цикл ($4 < 10$) и поэтому цикл *while* совершит ещё один свой очередной шаг-итерацию.

До тех пор, пока (*while*) условие входа в цикл выполняется, трудяга *while* честно запускает все операторы своего тела. Но как только ix увеличится до 12, «работник разорвёт трудовой договор», поскольку в этом случае $12 > 10$, а трудиться на таких условиях он согласия не давал.

После такого «расторжения договора» исполняется оператор, следующий за телом оператора *while*. В нашей функции *PuskClick* после тела оператора *while* нет никаких операторов, поэтому функция завершит свою работу выводом последних результатов: 10 и 200 (см. рис. 4.4).

Возможен вариант, при котором в теле цикла не будет изменяться условие входа в цикл. Что произойдёт тогда? В этом случае программа заикнется, цикл будет трудиться бесконечно.

Таким образом, цикл *while* исполняется только в случае, когда выполняется условие входа в цикл. В нашей программе – это выполнение условия о том, чтобы аргумент был равен или меньше x_Max . Каждый новый шаг цикла осуществляется только при выполнении условия входа в цикл. Поэтому этот цикл называют *циклом с предусловием*. Если условие входа в цикл не выполняется уже до первого шага, то цикл игнорируется и выполняется оператор, следующий за этим циклом. В теле цикла обязательно должен быть оператор, влияющий на условие входа в цикл.

Последний оператор тела цикла $ix = ix + dx$ в целях повышения быстродействия и компактности программы можно заменить оператором $ix += dx$ (подроб-

нее см. подраздел 3.3.5). Ясность программе такой и подобные ему операторы никак не добавляют, поэтому к ним лучше прибегать на заключительном этапе отладки программы, или же вообще воздержаться от их использования на первых шагах обучения программированию.

Приведём теперь запись цикла *while* в общем виде:

```
while (Условие входа в цикл)
{
    //Тело цикла
} //while
```

Если тело цикла состоит всего лишь из одного оператора, то фигурные скобки можно не ставить. Также как и в операторе *for*, для экстренного выхода из цикла *while* применяются оператор *break* и функция *Abort()*.

Вопросы для текущего самоконтроля

- ❑ Как будет работать функция *PuskClick* с циклом *while*, в случае замены условия $ix \leq x_Max$ на $ix \geq x_Max$? (на $ix == x_Max$; на $ix = x_Max$; на $ix != x_Max$; на 1, на -3.32 ?)
- ❑ Что будет выведено на экране функцией *PuskClick*, если в теле оператора *while* исключить оператор $ix = ix + dx$, если этот оператор заменить таким $ix = ix - dx$, если исключить оператор $++ij$; поменять местами $ix = ix + dx$; и $++ij$; если в конце цикла поставить оператор $ix = x_Max$?

4.4. Оператор цикла *do_while*

Также как и *while*, цикл *do_while* в первую очередь предназначен для решения задач, в которых нет возможности заранее определить число повторений цикла, однако он применим и для решения прежней учебной задачи, где число шагов цикла легко вычисляется до начала его работы. Ниже приведен фрагмент кода функции *PuskClick* в случае применения цикла *do_while*. Проект в этом случае будет называться *O_do_while*.

```
//Вычисляем ix и iy
ij= 1; ix=x_Min; //Начальные значения для входа в цикл
do
{
    Rezyltat->Cells[0][ij]= ix;
    iy= a*pow(ix, 2);
    Rezyltat->Cells[1][ij]= iy;
    ix+= dx;
    ij++;
} //do
while (ix<= x_Max); //Постусловие
} //PuskClick
```

Тело оператора цикла *do_while* находится между служебными словами *do* (делать) и *while*, оно полностью совпадает с телом оператора цикла *while*. В цикле *do_while* нет условия входа в цикл, однако имеется условие выхода из него. Такое условие полностью совпадает с условием входа в цикл *while* и стоит также после

ключевого слова *while*. Поскольку в цикле *do_while* условие расположено в конце тела цикла, поэтому этот цикл называется *циклом с постусловием*, он всегда делает первый свой шаг до проверки условия (или условий) продолжения цикла.

В общем случае цикл *do_while* записывается в виде:

```
do
{
    //Тело цикла
} //do
while (Условие продолжения цикла); //Постусловие
```

Если тело цикла состоит всего лишь из одного оператора, то фигурные скобки можно не ставить. Также как и в цикле *while*, в теле цикла *do_while* должны быть оператор или операторы, влияющие на условие продолжения цикла. Иначе программа зависнет. Для экстренного выхода из цикла используются оператор *break* и функция *Abort()*.

Вопросы для текущего самоконтроля

- ☐ Как будет работать программа *O_do_while*, в случае замены условия $ix \leq x_Max$ на $ix > x_Max$? (на $ix == x_Max$; на $ix = x_Max$; на 0; на 1; на -3.32 ?)
- ☐ Что будет выведено на экране программой *O_do_while*, если в теле оператора *do_while* исключить оператор $ix = ix + dx$, если этот оператор заменить таким $ix = ix - dx$?

4.5. Вложенные циклы

Проиллюстрируем применение вложенных циклов на простом примере. Требуется начертить график параболы $y = a \cdot x^2$ при нескольких значениях коэффициента a . Пусть параметр a изменяется в диапазоне от 2 до 6 с приращением $da = 2$, а область изменения аргумента x остаётся прежней (от 0 до 10 с шагом 2). Для выполнения такого задания, прежде всего, надлежит в заданном диапазоне x вычислить y для всех значений a . Иными словами – протабулировать функцию y для заданных условий.

Имеется два способа расчёта таких таблиц. Первый из них основывается на применении одной из программ: *O_for*, *O_while* или *O_do_while*. В выбранной программе можно несколько раз изменить значение параметра a . Конечно, такой путь возможен, однако при увеличении числа значений a он весьма утомителен и неразумен. Вложенные циклы решают эту и аналогичные задачи наиболее изящно: один цикл изменяет параметр a , а другой – переменную x .

Рассмотрим вначале вложенные циклы на примере оператора *for*. Собственно при помощи этого цикла и нужно решать эту задачу ведь число шагов в ней как по x , так и по a легко определяется. Программы с этим оператором цикла очень понятны, учащиеся при использовании этого цикла значительно меньше допускают ошибок, приводящих к заикливанию.

Экспериментально мы установили, что быстроедействие циклов *for*, *while* и *do_while* при одних и тех же условиях практически одинаково. Этот результат

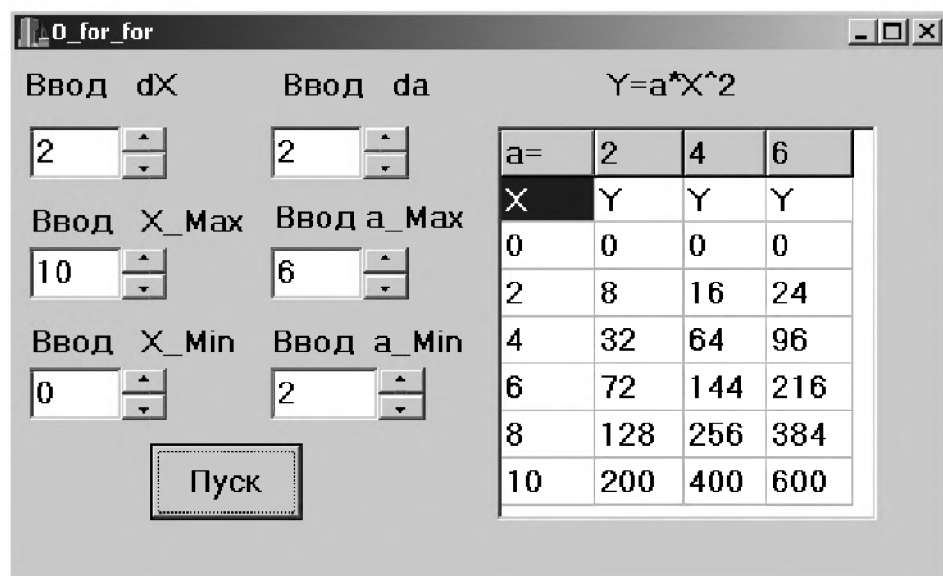


Рис. 4.5. Интерфейс приложения O_for_for

вполне объясним, ведь в любом из упомянутых операторов повтора *на каждом* шаге осуществляется проверка условия продолжения цикла. Поэтому *грамотный* выбор оператора цикла определяется *только* условием задачи, стремлением разработчика сделать программу наиболее ясной.

Для знакомства с работой вложенных циклов скопируйте в новую директорию *O_for_for* все файлы проекта *O_for*, переименуйте их, интерфейс измените так, как показано на рис. 4.5.

Вы видите, что на интерфейсе появились поля ввода шага изменения и граничных значений параметра *a* (*da*, *a_Max* и *a_Min*) с соответствующими заголовками этих полей. Во всех полях ввода имеются стартовые значения, указанные в формулировке нового варианта учебной задачи. Такие изменения интерфейса вы легко выполните самостоятельно по примеру предшествующих проектов. Обращаем внимание на то, что в *Инспекторе Объектов* в свойствах таблицы строк *ColCount* и *RowCount* можно не устанавливать соответственно число столбцов и строк. В приложении эти свойства настраиваются программно в функции *PyskClick*.

Для вывода увеличенного объёма новых данных следует расширить таблицу *Rezyltat* и дополнить её новыми заголовками. Из интерфейса вы видите, что на нулевой строке таблицы выведен заголовок *a=* (в нулевом столбце) и все значения этой величины, которая теперь является не константой, а переменной. Заголовки для столбцов *x* и *y* располагаются на *первой* строке (ранее они помещались на нулевой строке), а их расчётные значения размещены на строках с номерами от 2 до 7, в таблице имеется 8 строк и 4 столбца.

Приведенные пояснения помогут вам изучить модернизированный код функции *PyskClick*. Подробное объяснение работы этой функции приведено после её кода.

```
void __fastcall TO_for_for::PyskClick(TObject *Sender)
{
    const int dx= StrToInt(Znachen_dx->Text),
              x_Min= StrToInt(Znachen_X_Min->Text),
```

```

    x_Max= StrToInt(Znachen_X_Max->Text),
    n= (x_Max - x_Min)/dx + 1,
    da= StrToInt(Znachen_da->Text),
    a_Min= StrToInt(Znachen_a_Min->Text),
    a_Max= StrToInt(Znachen_a_Max->Text),
    na= (a_Max - a_Min)/da + 1;

//Устанавливаем число столбцов и строк таблицы строк
Rezyltat->ColCount= na + 1;
Rezyltat->RowCount= n + 1;
//Называем столбцы
Rezyltat->Cells[0][0]= "a=";
Rezyltat->Cells[0][1]= "X";
for (int ia= a_Min, ii= 1; ii<=na; ++ii)
{
    Rezyltat->Cells[ii][0]= ia;
    Rezyltat->Cells[ii][1]= "Y";
    for (int ij= 2, ix, iy; ij<= n + 1; ++ij)
    {
        ix= x_Min + (ij - 2)*dx;
        Rezyltat->Cells[0][ij]= ix;
        iy= ia*pow(ix, 2);
        Rezyltat->Cells[ii][ij]= iy;
    }//for_ij
    ia= ia + da;
} //for_ia
} //PyskClick

```

Тело функции, помимо операторов, предназначенных для вывода заголовков строки «a=» и столбца «X», состоит из циклов: по *ia* и по *ij*. В цикле по *ij* вы узнаете тело программы *O_for*, этот цикл (как и прежде) обеспечивает расчет аргумента *ix* и функции *iy* для выбранного значения *ia*. Однако в инициализирующей части его заголовка *for (int ij= 2, ix, iy; ij<= n+1; ++ij)* не только присваивается значение 2 переменной *ij* (почему 2?), но и объявляются *все* переменные (*ij, ix, iy*), которые используются *только в этом* цикле и нигде больше.

В указанном разделе инициализирующих операторов переменные могут объявляться только *одного* типа, поскольку все операторы этого раздела могут разделяться лишь запятыми. При объявлении переменных с разными типами пришлось бы их отделять друг от друга точкой с запятой, что противоречит синтаксису заголовка цикла *for*, согласно которому точкой с запятой разделяются *только* упомянутые три его раздела. Поэтому в случае объявления в первом блоке разнотипных переменных *Builder* сообщит об ошибке. Объявив же разнотипные переменные вне заголовка цикла, в его разделе инициализации всем переменным *можно* присваивать *разнотипные* значения.

После объявления переменных в разделе инициализации у них начинается *время жизни*, они *видимы* лишь в заголовке и теле этого оператора *for*, за телом время их жизни заканчивается, а память, занятая ими *освобождается* и может быть использована для размещения любых других переменных и объектов. Такое ограничение пространства видимости и времени жизни переменных является *огромным* достоинством *C++* и *C++Builder*. *ни один* другой язык высокого уровня не обладает таким удобным инструментом разработки программ. При необходи-

мости имеется возможность в функции *PyskClick* ввести новые переменные с таким же именем, чётко обозначив фигурными скобками область их действия (видимости) и время жизни.

Что произойдёт, если не исключить объявление переменных (*ij*, *ix*, *iy*) в начале функции *PyskClick*? То есть оставить без изменений раздел объявления переменных из проекта-болванки, использованного в качестве заготовки настоящего приложения. Такой поступок не приведёт к ошибке. *Builder* лишь предупредит разработчика программы о том, что упомянутые переменные объявлены, однако нигде не используются. Эти переменные являются *глобальными* для *всего* тела функции *PyskClick*, их можно использовать (они видимы) *в любом месте* тела функции, *кроме* логических блоков, в которых имеются *локальные одноимённые* переменные. Поэтому локальные переменные, введенные внутри логического блока, делают недоступными в этом блоке глобальные переменные с тем же именем. Имеется средство сделать видимыми в таких логических блоках и одноимённые глобальные переменные, однако о нём расскажем лишь на 6 уроке.

Не рекомендуется использовать совершенно одинаковые идентификаторы переменных программы, даже если они видимы в разных её частях и относятся к тем же по характеру величинам. Не следует бездумно эксплуатировать богатые возможности языка программирования, лучше дайте таким переменным близкие названия. Применяйте следующее правило: чем более локальный блок, чем меньше область его видимости – тем меньше литер в идентификаторе переменной, которая в нём объявлена.

Если переменные (или объекты) функции вводятся в начале её тела, то для этой функции они называются *глобальными* и могут использоваться в любом месте её тела. Если же переменные (или объекты) объявлены внутри логического блока функции, то для функции они именуются *локальными*, но являются глобальными для блока их ввода и всех логических блоков внутри него. Локальные переменные вне своего логического блока невидимы, за блоком время их жизни заканчивается.

В управляющем разделе заголовка внутреннего цикла использовано условие $ij \leq n + 1$, а не применяемое ранее условие $ij \leq n$. Это обусловлено тем обстоятельством, что строки вывода данных сдвинуты на одну строку из-за необходимости вывода на первой строке заголовков столбцов для *X* и *Y*.

Цикл *for*, предназначенный для изменения *ia*, является внешним по отношению к циклу *for*, служащему для расчёта (и вывода) *ix* и *iy*. В заголовке внешнего цикла *for (int ia = a_Min, ii = 1; ii <= na; ++ii)* также осуществляется не только инициализация переменных *ia* и *ii*, но и происходит их объявление. В результате эти переменные видимы в заголовке и теле рассматриваемого цикла, в заголовке и в теле цикла по *ij*, который, как матрёшка вставлен внутрь цикла по *ia*.

Переменная *ii*, изменяется от 1 до *na* (число шагов по *ia*), во внутреннем цикле она управляет номером выводимого столбца с очередным вариантом расчёта *iy*. На первом шаге этого цикла, при *ii* = 1, значение *iy* рассчитывается для *a_Min*, и выводится в первом столбце таблицы. Последующие варианты расчёта *iy* выводятся во втором и третьем столбцах.

После завершения расчётов ix и iy при $ii = 1$ (для a_Min) переменная ia получает приращение на величину шага своего изменения $ia = ia + da$. Теперь расчёт ix и iy выполняется для нового значения ia . В нашем примере третий цикл по ia является последним, он рассчитывает и выводит величину ix и iy для максимального значения ia .

В приведенном примере одни и те же значения ix рассчитываются при каждом прогоне внутреннего цикла, при каждом его запуске они размещаются в одном и том же (нулевом) столбце. Повторяем, номер столбца с очередным вариантом расчёта iy *сдвигается* на единицу после *каждого* пробега внутреннего цикла. Для вывода заголовка этого столбца предназначен оператор `Rezyltat->Cells[ii][1] = "Y";`. Он находится перед внутренним циклом `for`. Рядом с упомянутым оператором расположен оператор `Rezyltat->Cells[i][0] = ia;`, предназначенный для вывода на нулевой строке таблицы значений ia для каждого варианта расчёта iy .

В данном примере оператор $ia = ia + da$, используемый для увеличения значения ia , можно записать в более компактной форме $ia += da$.

Вложенные циклы могут состоять как из операторов цикла *одного* выбранного типа, так и операторов циклов `for`, `while`, `do_while` в любой последовательности. Число вложений циклов не ограничено. Вариант последней задачи с применением циклов `while` и `do_while` в функции `PyskClick` может иметь такой вид:

```
void __fastcall TO_while_do_while::PyskClick(TObject *Sender)
{
    const int dx= StrToInt(Znachen_dx->Text),
              x_Min= StrToInt(Znachen_X_Min->Text),
              x_Max= StrToInt(Znachen_X_Max->Text),
              n= (x_Max - x_Min)/dx + 1,
              da= StrToInt(Znachen_da->Text),
              a_Min= StrToInt(Znachen_a_Min->Text),
              a_Max= StrToInt(Znachen_a_Max->Text),
              na= (a_Max - a_Min)/da + 1;

    //Устанавливаем число столбцов и строк таблицы строк
    Rezyltat->ColCount= na + 1;
    Rezyltat->RowCount= n + 1;

    //Называем столбцы
    Rezyltat->Cells[0][0]= "a=";
    Rezyltat->Cells[0][1]= "X";

    //Вычисляем ix и iy
    int ia= a_Min, ii= 1;
    while (ii<= na)
    {
        Rezyltat->Cells[ii][0]= ia;
        Rezyltat->Cells[ii][1]= "Y";
        int ij= 2, ix, iy;
        do
        {
            ix= x_Min + (ij - 2)*dx;
            Rezyltat->Cells[0][ij]= ix;
```

```
        iy= ia*pow(ix, 2);
        Rezyltat->Cells[ii][ij]= iy;
        ++ij;
    }//do
    while (ij<= n + 1);
    ia+= da;
    ++ii;
} //while
} //PyskClick
```

В этом варианте переменные *ia* и *ii* являются глобальными переменными функции *PyskClick*, а *ij*, *ix* и *iy* – локальными для этой функции, но глобальными для внутреннего цикла *do_while*.

Переменные *ix*, *iy* и *ia* в двух последних вариантах функциях *PyskClick* имеют целочисленный тип *int*. Это обусловлено тем обстоятельством, что в нашей задаче все их значения являются целыми числами и не выходят за границы диапазона выбранного типа.

Если расширить границы определения аргумента, и увеличить показатель степени при *ix*, то может оказаться, что *iy* уже не удовлетворяет типу *int*. В этом случае для хранения результатов расчёта можно прибегнуть к применению вещественных типов *float*, *double* или *long double* (см. раздел 3.2), имеющих значительно больший интервал допустимых значений.

Выбрать вещественный тип мы обязаны также в случае вещественного шага, вещественной начальной границы аргумента или вещественного показателя степени, словом во всех случаях, когда переменные окажутся вещественными либо выйдут за границы целочисленного типа.

Таким образом, соответствие ожидаемых значений переменных их типам необходимо *всегда* тщательно *проверять*. Прочитав эту рекомендацию, прагматичный читатель может заявить: Да я всем своим переменным назначу тип *float* или даже *long double* и у меня не будет никаких проблем с несоответствием типа. Зачем же мучить себя арифметическими упражнениями?

Отдельные программисты именно так и поступают. Но этот путь, как вы уже знаете, ведёт к незначительному использованию оперативной памяти и существенному снижению быстродействия программ. Поэтому это плохой путь и его следует всячески избегать в своей практике. И так следует поступать не только в связи с указанными причинами.

Дело в том, что помимо упомянутых «преимуществ» «рациональный» путь уменьшает ещё и устойчивость программы к ошибкам. Предположим, вы правильно оценили диапазон изменения вашей целочисленной переменной и выбрали ей соответствующий тип данных. Но на определённом шаге выполнения программы, из-за каких то ошибок, значение переменной выходит за границу назначенного ей типа. В этом случае в других языках программирования заботливый компилятор (или интерпретатор) остановит программу и подсветкой выделит ту переменную, в которой произошло, например, переполнение. Вы, конечно, огорчитесь, однако легко устраните ошибку.

Но в *C++Builder* (и в *C++*) *нет таких* (и многих других) проверок на соответ-

ствие границам числовых типов. Именно поэтому программы, написанные на этом языке, имеют наибольшее быстродействие, ведь в них время на эти проверки не расходуется. Поэтому программист *должен* взять на себя *всю* ответственность за такой контроль. Для повышения надёжности работы программ следует контролировать интервал изменения *всех*, или, по крайней мере, самых основных переменных. Это можно делать, например, при помощи оператора выбора *if*. После превышения значения переменной заданной границы можно организовать прерывание работы программы с выводом соответствующего сообщения. Повторяем, что такой контроль рекомендуется выполнять не взирая на увеличение объёма кода и уменьшение быстродействия программы: надёжность – прежде всего!

В противном же случае, когда с запасом выбрать тип данных, не расставлять капканы для отлова возможных ошибок, то в упомянутой выше ситуации останова программы *не произойдёт*, и в итоге будут ошибочные результаты вычислений. Это особенно опасно в случаях, когда такие результаты оказываются правдоподобными, поскольку они приводят к *ложным* выводам, ошибочность которых трудно определяется.

Вопросы для самоконтроля

- ☐ Зачем нужны циклы?
- ☐ Когда используется цикл *for*?
- ☐ При каких условиях применяются циклы *while* и *do_while*?
- ☐ Сколько раз запускается внутренний цикл в двух вложенных циклах?

4.6. Задачи для программирования

Задача 4.1. Разработайте программу расчета функции

$$y = 4ax^2 + 3,7b + c,$$

где $c = 2,7$;

$$a \in [1; 2], da = 0,5;$$

$$b \in [1; 2], db = 0,5;$$

$$x \in [0; 1], dx = 0,2.$$

Задача 4.2. Рассчитайте значение функции

$$y = \begin{cases} \sin(x + 0,75), & \text{если } |x| \leq 5; \\ x^4 + 2x^2 \cos x, & \text{если } |x| > 5, \end{cases}$$

где $x \in [-10, 10]$, $\Delta x = 1$.

Задача 4.3. Рассчитайте высоту подъема H тела, брошенного под углом α к горизонту с начальной скоростью V , для $\alpha \in [0 \text{ град}; 80 \text{ град}]$, $\Delta\alpha = 10 \text{ град}$ и $V \in [20 \text{ м/с}; 100 \text{ м/с}]$, $\Delta V = 20 \text{ м/с}$.

4.7. Варианты решений задач

Интерфейс приложения *Задача 4.1* показан на рис. 4.6. Функция по обработке нажатия клавиши *Пуск* приведена ниже.

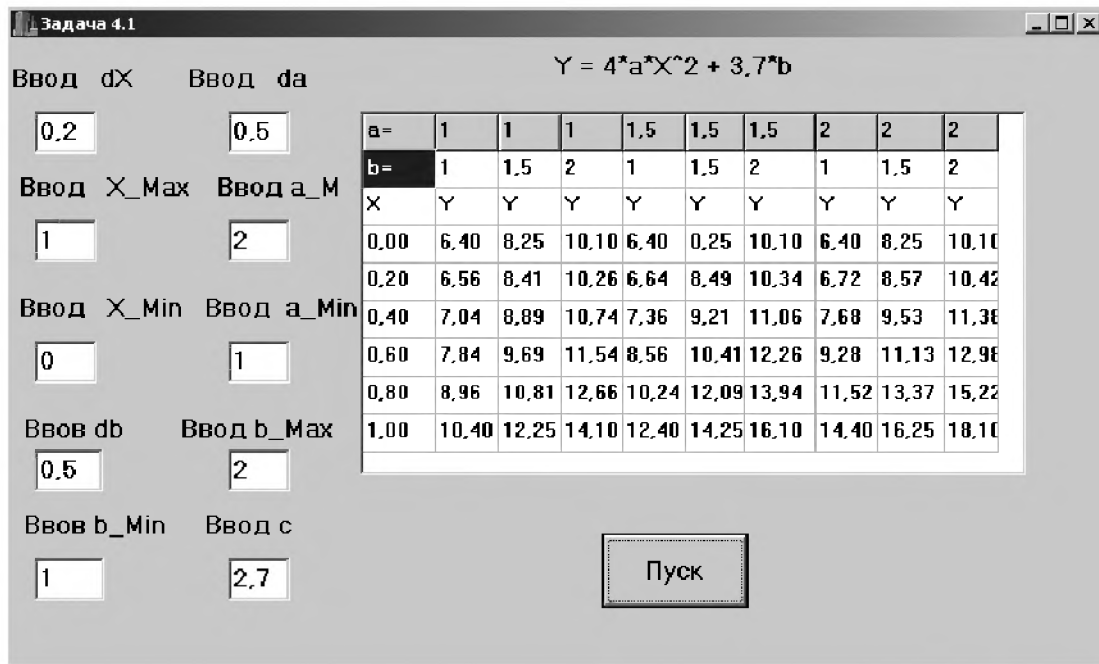


Рис. 4.6. Интерфейс приложения *Задача 4.1*

```
void __fastcall TZadacha_4_1::PyskClick(TObject *Sender)
{
    const float dx= StrToFloat(Vvod_dx->Text),
                x_Min= StrToFloat(Vvod_X_Min->Text),
                x_Max= StrToFloat(Vvod_X_Max->Text),

                da= StrToFloat(Vvod_da->Text),
                a_Min= StrToFloat(Vvod_a_Min->Text),
                a_Max= StrToFloat(Vvod_a_Max->Text),

                db= StrToFloat(Vvod_db->Text),
                b_Min= StrToFloat(Vvod_b_Min->Text),
                b_Max= StrToFloat(Vvod_b_Max->Text),
                c= StrToFloat(Vvod_c->Text);

    //Называем столбцы
    Rezyltat->Cells[0][0]= "a=";
    Rezyltat->Cells[0][1]= "b=";
    Rezyltat->Cells[0][2]= "X";
    int ii= 1, ik= 1;
    for (float fa= a_Min; fa<= a_Max; fa+= da, ii++)
    {
        for (float fb= b_Min; fb<= b_Max; fb+= db, ik++)
        {
            Rezyltat->Cells[ik][0]= fa;
            Rezyltat->Cells[ik][1]= fb;
            int ij= 3;
```

```

for (float fx= x_Min, fy; fx<= x_Max; fx+= dx, ij++)
{
    Rezyltat->Cells[0][ij]=
        FloatToStrF(fx, ffFixed, 10, 2);
    fy = 4*fa*pow(fx, 2) + 3.7*fb + c;
    Rezyltat->Cells[ik][2]= "Y";
    Rezyltat->Cells[ik][ij]= FloatToStrF(fy, ffFixed, 10, 2);
} //for_fx
} // for_fb
} //for_fa
} // PyskClick

```

Интерфейс приложения *Задача 4.2* показан на рис. 4.7. Текст функции *PyskClick* по обработке нажатия клавиши *Пуск* приведен ниже рисунка.

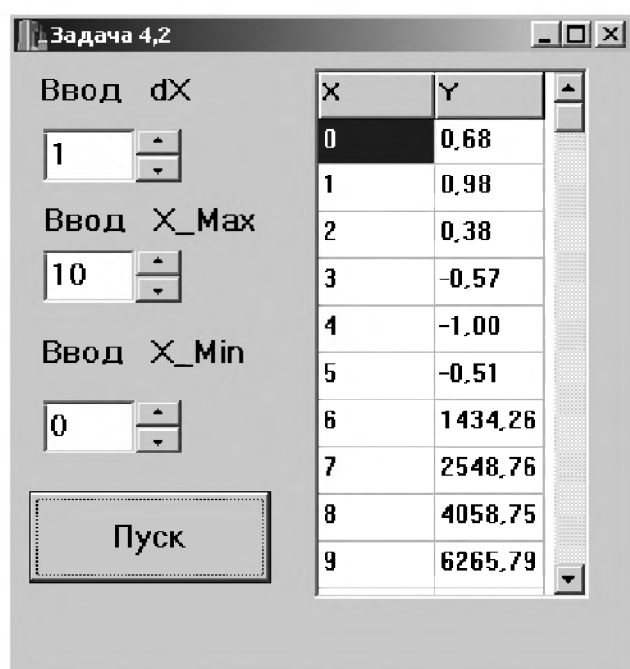


Рис. 4.7. Интерфейс приложения *Задача 4.2*

```

void __fastcall TZadacha_4_2::PyskClick(TObject *Sender)
{
    const int dx= StrToInt(Vvod_dx->Text),
              x_Min= StrToInt(Vvod_X_Min->Text),
              x_Max= StrToInt(Vvod_X_Max->Text);
    int ij=1, ix;
    double dy;
    //Называем столбцы
    Rezyltat->Cells[0][0]= "X";
    Rezyltat->Cells[1][0]= "Y";

    for (ix= x_Min; ix<= x_Max; ix+= dx, ++ij)
    {
        Rezyltat->Cells[0][ij]= ix;
        if (abs(ix)> 5)
            dy= pow(ix, 4) + 4*pow(ix, 2)*cos(ix);
        else

```

```

dy= sin(ix + 0.75);
Rezyltat->Cells[1][ij]= FloatToStrF(dy, ffFixed, 10, 2);
} // for_ix
} // PyskClick

```

Интерфейс приложения *Задача 4.3* показан на рис. 4.8. Функция по обработке нажатия клавиши *Пуск* приведена ниже рисунка.

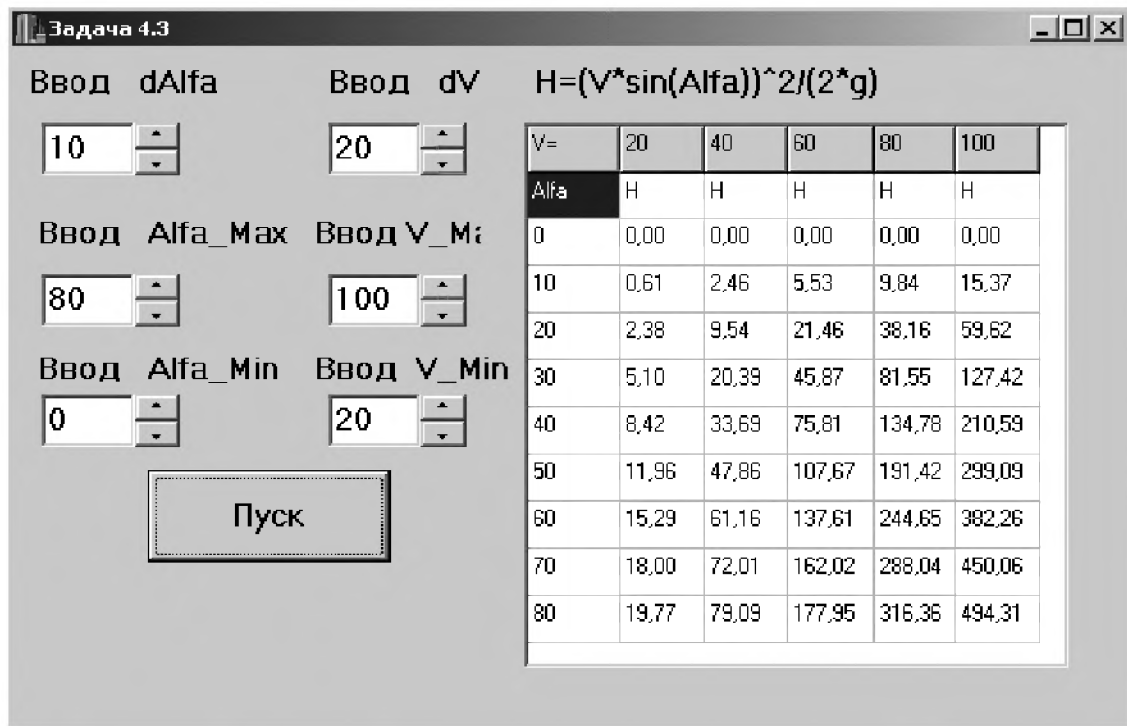


Рис. 4.8. Интерфейс приложения *Задача 4.3*

```

void __fastcall TZadacha_4_3::PyskClick(TObject *Sender)
{
    const int dV= StrToInt(Vvod_dV->Text),
              V_Min= StrToInt(Vvod_V_Min->Text),
              V_Max= StrToInt(Vvod_V_Max->Text),
              dAlfa= StrToInt(Vvod_dAlfa->Text),
              Alfa_Min= StrToInt(Vvod_Alfa_Min->Text),
              Alfa_Max= StrToInt(Vvod_Alfa_Max->Text);
    const float Pi=3.14159, g= 9.81;
    float fAlfa_Rad, fH;
    //Называем столбцы
    Rezyltat->Cells[0][0]= "V=";
    Rezyltat->Cells[0][1]= "Alfa";
    //Вычисляем iV, iAlfa и fH - скорость, угол и
    //высоту подъёма
    for (int ii= 1, iV= V_Min; iV<= V_Max; iV+= dV, ++ii)
    {
        Rezyltat->Cells[ii][0]= iV;
        Rezyltat->Cells[ii][1]= "H";
        for (int iAlfa=Alfa_Min, ij= 2;
              iAlfa<= Alfa_Max; iAlfa+= dAlfa, ++ij)
        {

```

```
fAlfa_Rad= iAlfa*Pi/180;
Rezyltat->Cells[0][ij]= iAlfa;
fH= pow(iV*sin(fAlfa_Rad), 2)/(2*g);
Rezyltat->Cells[ii][ij]= FloatToStrF(fH, ffFixed, 10, 2);
} // for_iAlfa
} // for_ii
} // PyskClick
```


Урок 5. Графики зависимостей

На прошлом уроке мы научились рассчитывать функциональные зависимости и представлять результаты расчётов в таблицах. Однако анализ табличных данных менее нагляден, чем анализ графиков. По этой причине в программах табулированные результаты расчётов обычно помещались в текстовые файлы. Затем графики достаточно высокого качества строились в каком-либо графическом редакторе, например, в редакторе инженерных и научных графиков *SigmaPlot*. Этот редактор позволяет также производить обработку и статистический анализ исходных данных.


Языки программирования высокого уровня, безусловно, также позволяют строить графики зависимостей. Однако при большом объёме программистских работ качество графиков весьма низкое, их практически невозможно включать в научные статьи и отчёты без привлечения специального графического редактора.

С появлением *C++Builder* (и *Delphi*) ситуация радикально изменилась. Возможности этого продукта (графические в частности) поражают не только высоким качеством результатов, но и лёгкостью, с какой они достигаются. При этом имеется широкий набор вариантов реализации тех или иных задач. Например, графики функций можно построить с использованием одного из *четырёх* визуальных компонентов.

Далее рассмотрим визуальный компонент *Chart* (диаграмма). Среди других графических компонентов он наиболее мощный по своим возможностям и вместе с тем очень удобен для первого знакомства.

5.1. Построение одномерных зависимостей

Компонент *Chart* позволяет выводить различные диаграммы и графики. Применим его для построения графика параболы $y = a \cdot x^2$ в интервале x от 0 до 10 с шагом $dx = 2$ при $a = 2$.

Ниже иллюстрируется один из вариантов последовательностей шагов построения этого графика. Компонент *Chart* выглядит вот так: , находится на вкладке *Additional*. После размещения на форме объекта *Chart1* (имя по умолчанию не изменяйте) вызовите *Редактор Диаграмм*: сделайте двойной щелчок по объекту *Chart1* (см. рис. 5.1). Вызов редактора также осуществляется при помощи контекстного (подручного) меню объекта *Chart1* и его пункта *Edit Chart*. Далее окно редактора расположите рядом с объектом *Chart1*, чтобы редактор и объект одновременно были видны на экране.

Поскольку имя объекта – *Chart1*, в заголовке редактора указано «*Редактирование Chart1*» (*Editing Chart1*). В окне редактора имеются страницы (1) и заклад-

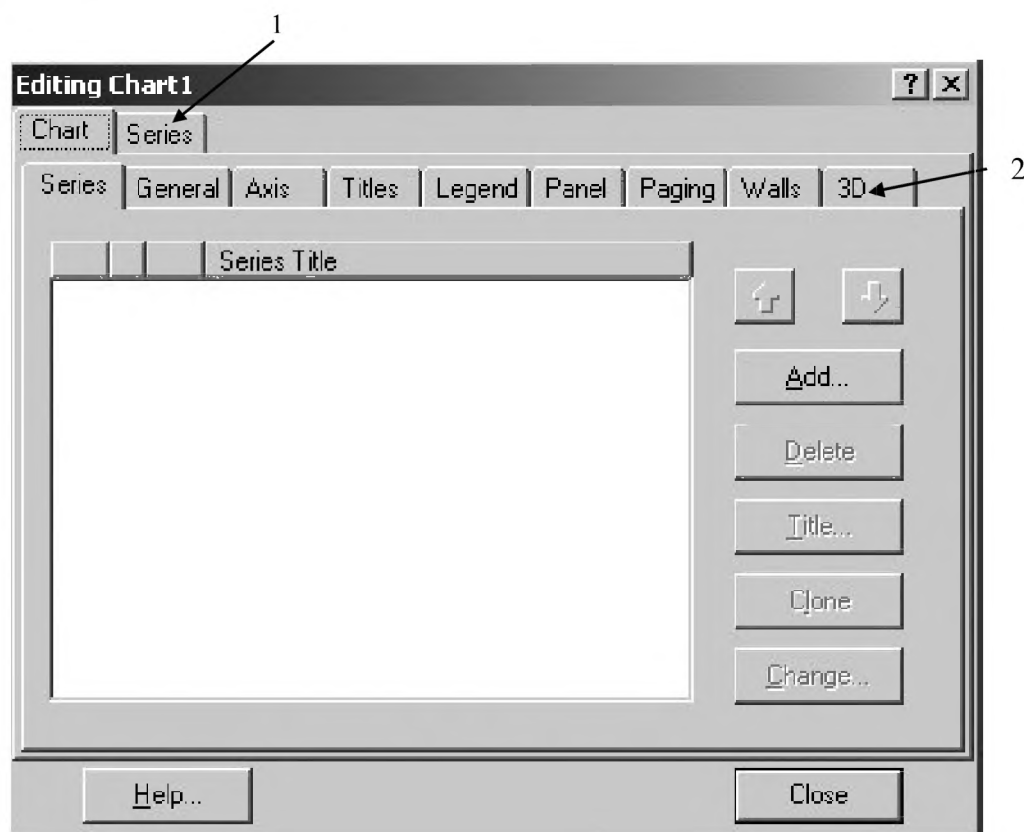


Рис. 5.1. Редактор Диаграмм, 1–страницы, 2–закладки

ки (2). При его запуске открывается страница *Chart* и закладка *Series* (на русском языке эту закладку лучше назвать *график*, здесь и ниже часто приводятся не переводы английских слов, а более ясные русифицированные аналоги служебных слов редактора). На этой закладке нажмите кнопку *Add* (добавить кривую графика). В результате появится окно с шаблонами трёхмерных графиков и диаграмм, показанное на рис. 5.2.

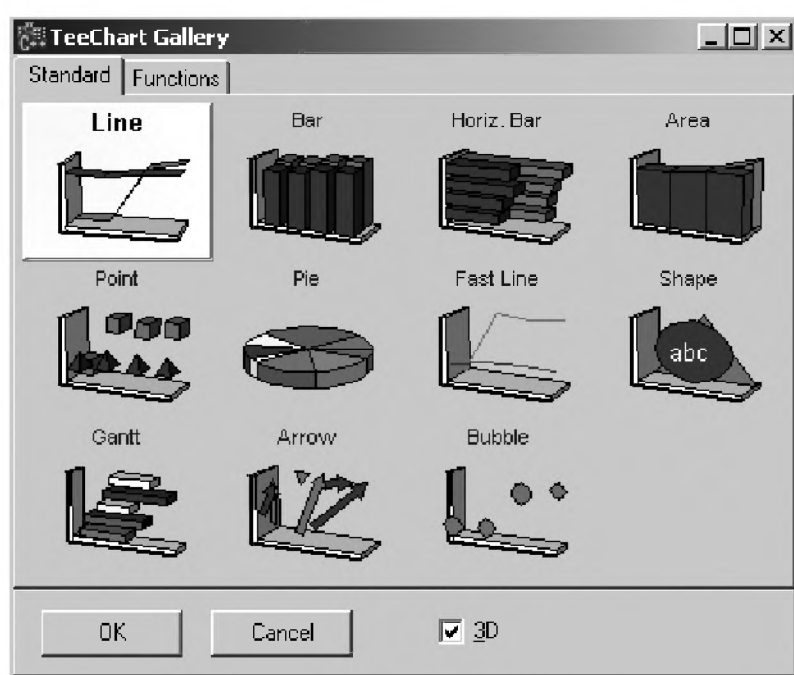


Рис. 5.2. Шаблоны трёхмерных графиков и диаграмм

Для решения учебной задачи трёхмерные шаблоны не нужны. Поэтому снимите птичку в поле *3D* этого окна. После этого на экране покажется окно с шаблонами *одномерных* графиков (см. рис. 5.3). В этом окне для наших целей наиболее подходит шаблон, расположенный в первом ряду первым слева. В этом шаблоне точки на графике соединяются отрезками *прямых* линий, поэтому шаблон называется *Line* (прямая линия).

После двойного щелчка по выбранному шаблону в окне *Редактора Диаграмм*, в его основном окне с именем *Series Title* (наименование графика), окажется стро-

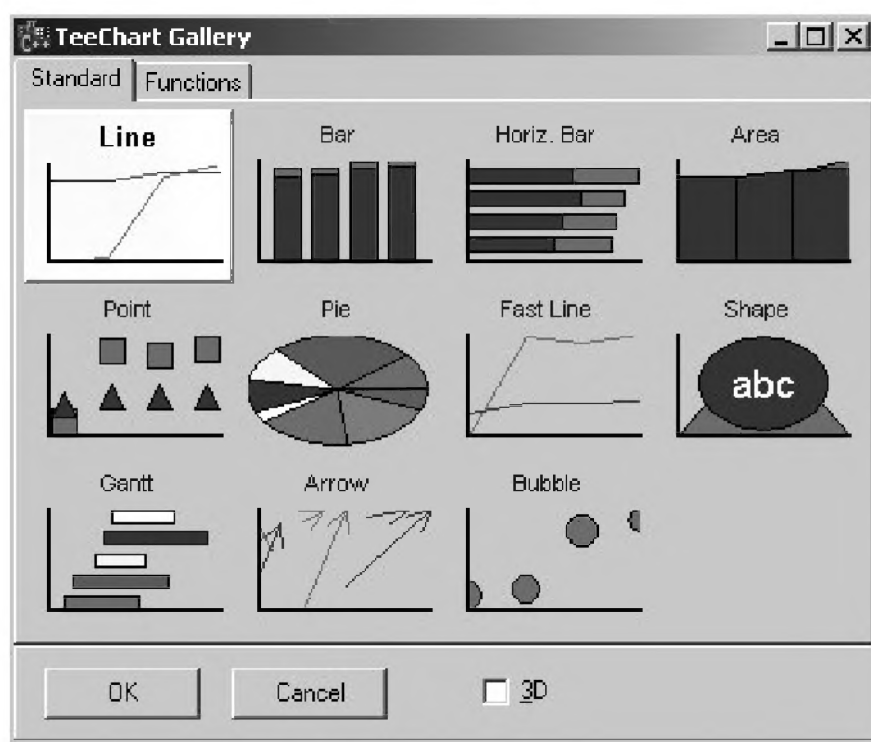


Рис. 5.3. Шаблоны для одномерных графиков и диаграмм

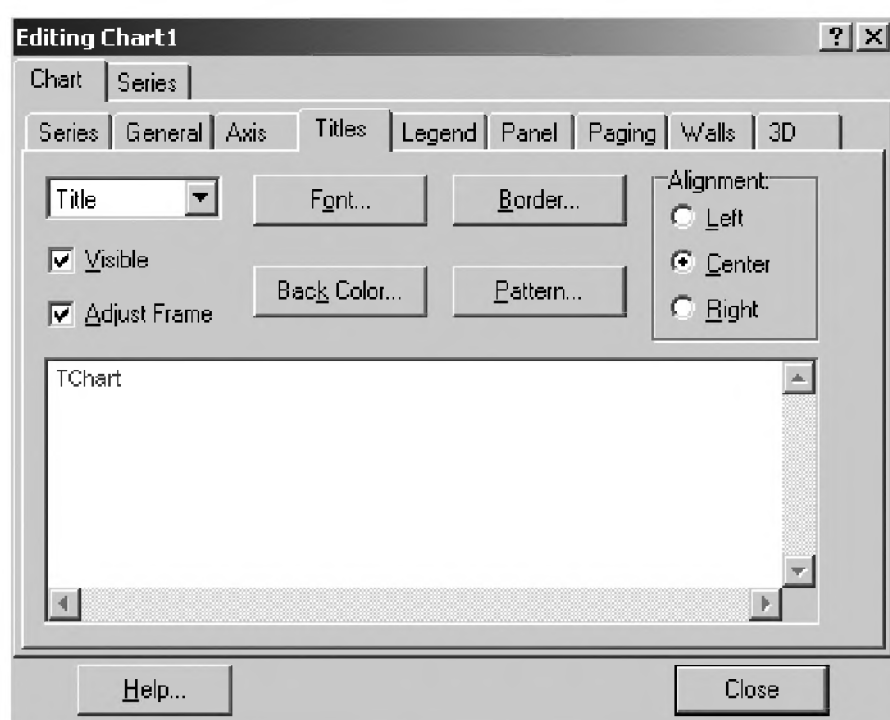


Рис. 5.4. Оформление названия графика

ка *Series1* (График1) с красным отрезком, иллюстрирующим предложенный редактором цвет и толщину линии графика. Впереди названия *Series1* имеются ещё иконка графика и окошко, предназначенное для вывода или скрытия кривой графика. При этом начальный вид объекта *Chart1* стал другим – на нём появился образец графика, предназначенный лишь для оперативной иллюстрации устанавливаемых параметров (или опций) графика. Цвет и толщина кривой графика совпадают с образцами, приведенными в окне *Series Title*.

По умолчанию *Редактор Диаграмм* назвал график именем типа используемого компонента (*TChart*). Ясно, что это имя лучше заменить более подходящим вариантом. С этой целью на прежней странице *Chart* выведите закладку *Titles* (оформление названия графика). Она показана на рис. 5.4. Далее, вместо исходного названия введите текст: «График функции $y=a*x^2$ ». Ход набора текста сразу же отражается в названии графика-примера, выведенного сверху графика.

При помощи радиокнопок *Left*, *Right*, *Center* раздела *Alignment* (выравнивание) имеется возможность расположить название соответственно возле левого, правого краёв, либо по центру (последняя опция установлена по умолчанию). С использованием раскрывающегося списка можно название графика (или какой-то другой текст) вывести также и внизу графика. Для этого в упомянутом списке выберите значение *Foot* (внизу). Текст можно вывести одновременно над и под графиком, только вверху или только внизу.

При помощи специальных кнопок выбирается вид и стиль шрифта (*Font*), цвет и толщина рамки вокруг текста (*Border*), цвет фона текста (*Back Color*), а также цвет и вид штриховки заднего плана текста (*Pattern*). После установки этих опций (можете оставить их по умолчанию) и закрытия *Редактора Диаграмм* вы увидите образец графика (например, таким, как на рис. 5.5).

Введите новое название *График* для заголовка формы и присвойте ей имя *Grafik*. Справа от графика на рис. 5.5 в прямоугольной рамке приведены обозна-

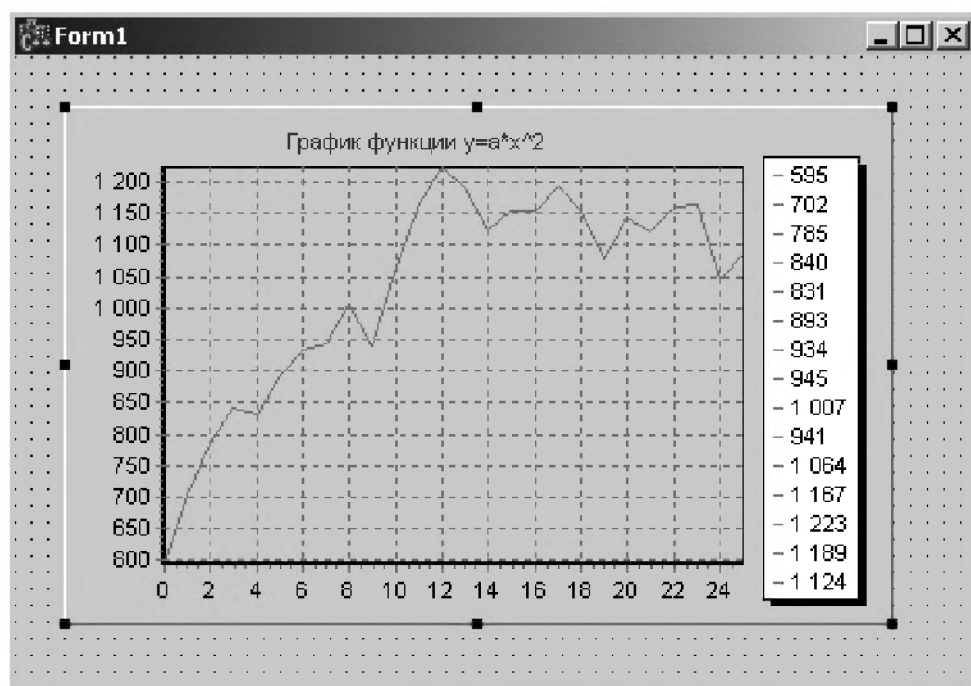


Рис. 5.5. Вид графика после замены его названия

чения для всех значений функции. В нашем примере такие обозначения не имеют смысла, они понадобятся лишь при построении *нескольких* кривых на графике (см. раздел 5.2). Поэтому сейчас уберём эти обозначения с экрана. Для этого необходимо вывести показанную на рис 5.6 закладку *Legend* (обозначения) и снять птичку в окне *Visible* (вывод).

Обозначения осей X и Y выводятся с использованием закладки *Axis* (ось) и её вкладки *Title* (название оси), показанных на рис 5.7. При помощи радиокнопок *Left* (слева), *Right* (справа), *Top* (сверху), *Bottom* (снизу), *Depth* (глубина) указы-

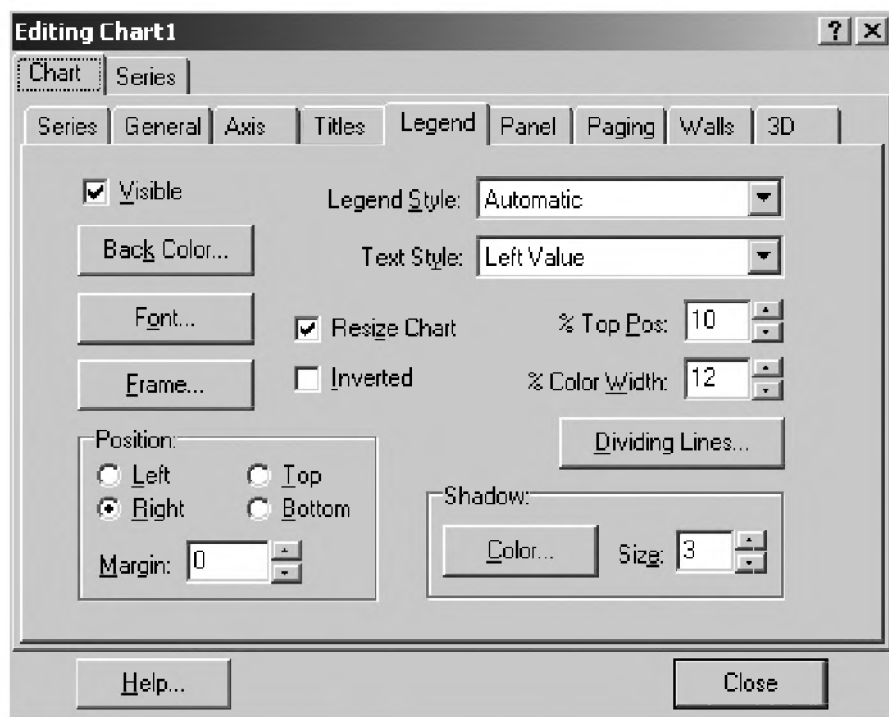


Рис. 5.6. Оформление обозначений графика, их вывод и устранение

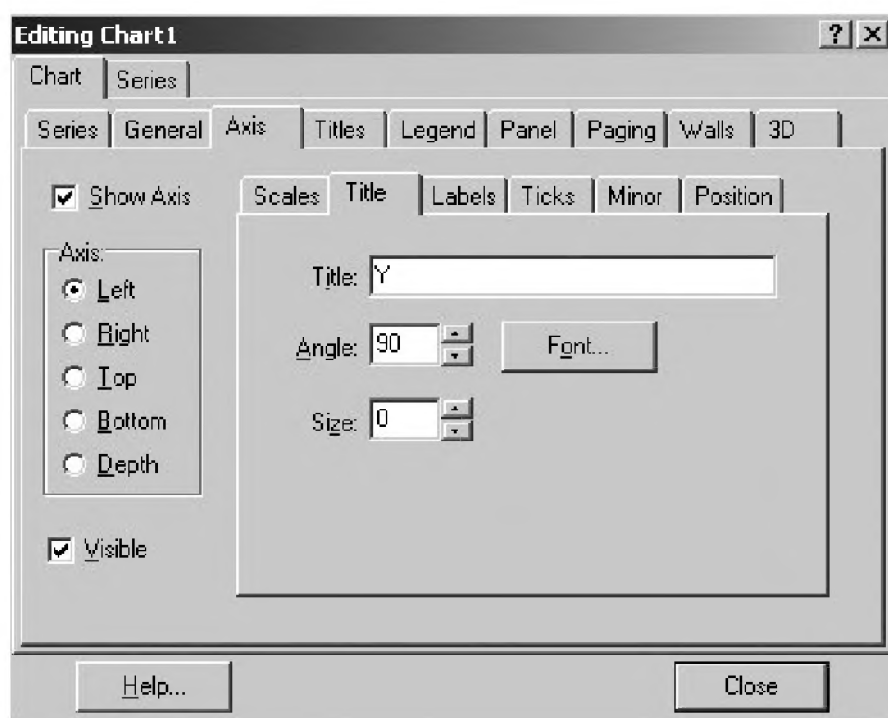


Рис. 5.7. Ввод названий для осей

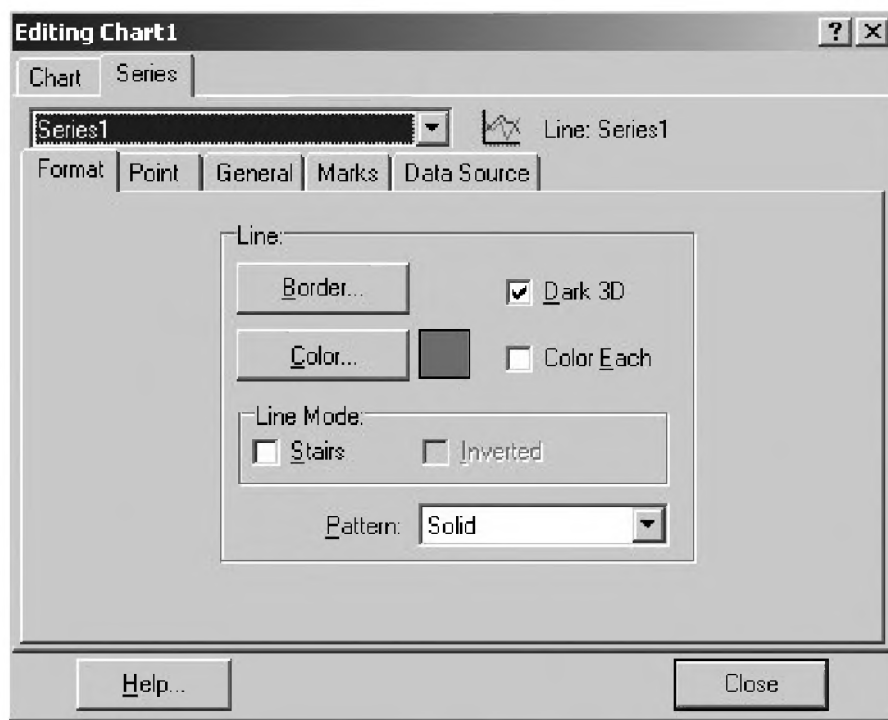


Рис. 5.8. Выбор параметров кривой графика

вается оформляемая ось. В поле ввода *Title* (название оси) набирается её имя (в нашем примере это имена *X* и *Y*). При помощи кнопок *Angle* (направление текста), *Font* (шрифт), *Size* (размер) в поле перед (ниже, выше, после) осью устанавливается пространственная ориентация названия, размер, стиль и вид шрифта, величина отступа от края окна объекта названия выбранной оси. При помощи окна *Visible* (вывод на экран) выводится (или убирается) на экран название оси.

Для оформления кривой графика выведите страницу *Series* (кривая) и её закладку *Format* (форматирование), они показаны на рис. 5.8. С использованием кнопок *Border* (вид кривой) и *Color* (цвет) выбираются вид кривой (сплошная, штриховая, точечная и др.) и её цвет. Окно *Color Each* (цветные отрезки) позволяет вывести разными цветами все отрезки кривой. Это окно и окно *Stairs* (ступеньки, в этом режиме точки графика соединяются ступеньками, как в диаграммах и гистограммах) оставьте без птички, в нашем примере эти режимы совершенно не нужны.

Здесь и далее мы не будем разъяснять назначение абсолютно всех кнопок и полей ввода, поскольку эти подробности отвлекают от изучения продукта «с высоты птичьего полёта», приучите себя исследовать назначение неизвестных вам элементов методом проб. Следует отметить, что реализация возможностей *C++Builder* осуществляется стандартными для *Windows* средствами и в большинстве случаев назначения элементов интерфейса интуитивно понятны.

На форме (с именем *Grafik*) разместите таблицу строк *Rezyltat* и кнопку *Pysk*. Поля ввода и вывода параметров в настоящем приложении для уменьшения кода программы использовать не будем (в реальных проектах так поступать нежелательно). В функцию *PyskClik* внесите следующий текст.

```
void __fastcall TGrafik::PyskClick(TObject *Sender)
{
```

```
const int a= 2, dx= 2, x_Min= 0, n= 6;
int ix, iy;

Series1->Clear(); //Очистка графика, можно не использовать
//Вычисляем ix и iy
for (int ij= 1; ij<= n; ++ij)
{
    ix= x_Min+ (ij - 1)*dx;
    Rezyltat->Cells[0][ij]= ix;
    iy= a*pow(ix, 2);
    Rezyltat->Cells[1][ij]= iy;
    Series1->AddXY(ix, iy, "", clRed);
} //for_ij
} //PyskClick
```

Функция *Clear()* предназначена для стирания графического изображения в объекте типа *TSeries*, в данном приложении её вызов излишен (используется лишь для иллюстрации её вызова), поскольку после порождения объекта *Series1* в нём ещё не выводился график.

Четыре строки тела цикла *for* (см. листинг, приведенный выше) перекочевали с прошлого урока. Разъясним назначение пятой строки:

```
Series1->AddXY(ix, iy, "", clRed);
```

Функция *AddXY* позволяет добавлять новую точку графика в объекте *Series1*. У неё четыре параметра. Первые два параметра определяют координаты (абсциссу и ординату) точки графика. Третий параметр – текстовая константа, является названием точки: всё, что набирается в двойных кавычках, выводится в качестве надписи у соответствующей абсциссы. Вывод названия точек в разрабатываемом приложении не нужен, поэтому на месте третьего параметра поставлена пустая строка. Четвёртый параметр – это выбранный вами цвет кривой. Он может быть красным *clRed* (*color Red*), синим (*clBlue*) и любым другим из цветов, указанных в справке по этому параметру (жмите клавишу *F1*).

Внимательный читатель может заметить, что в функции *PyskClick* пропущены операторы, предназначенные для вывода названий столбцов *X* и *Y* на нулевой строке таблицы строк *Rezyltat*. Эти операторы будем теперь запускать специальной функцией в ходе запуска приложения.

На вкладке *События* перечисляются все события, которые могут происходить с выделенным визуальным объектом формы и самой формой. Например, событием является нажатие командной кнопки, смена данных в поле ввода и др. *Порождение формы* (запуск программы, появление её интерфейса) – это также событие. Расширим опыт использования событий для разработки приложений.

Сделаем так, чтобы в начале работы приложения, после появления его интерфейса (со всеми визуальными объектами) выводились упомянутые выше названия столбцов. Для реализации этой цели выделите форму, перейдите в *Инспектор Объектов* на вкладку *События* и в поле ввода события *OnCreate* (при порождении) сделайте двойной щелчок мышью. После этого появится *Редактор Кода*, где ниже функции *PyskClick* будет сгенерирована заготовка функции

*TGrafik::FormCreate(TObject *Sender)*. Эта заготовка порождается также двойным щелчком по форме. Внутри пустого тела упомянутой функции впишите прежние операторы:


```
void __fastcall TGrafik::FormCreate(TObject *Sender)
{
    //Называем столбцы
    Rezyltat->Cells[0][0] = "X";
    Rezyltat->Cells[1][0] = "Y";
} //FormCreate
```

Поскольку функция *FormCreate* вызывается всякий раз при порождении формы (интерфейса), поэтому после запуска программы столбцы таблицы строк будут с соответствующими заголовками.

Код, обуславливающий вывод названия столбцов, *вместо* функции *FormCreate* можно разместить в теле конструктора – функции *TGrafik*. Его заготовка генерируется автоматически в файле реализации и выглядит вот так:

```
__fastcall TGraf::TGraf(TComponent* Owner)
: TForm(Owner)
{
}
```

После запуска приложения и нажатия командной кнопки интерфейс программы может выглядеть, например так, как показано на рис. 5.9.

Заголовок командной кнопки набран строчными буквами чёрного цвета, применён шрифт *Sans Serif* с размером в 12 пунктов и полужирным начертанием. Оформление текста заголовков осуществляется в *Инспекторе Объектов* при помощи свойства *Font* (шрифт) выделенного объекта интерфейса (если у объекта имеется заголовок). Следует дважды щёлкнуть в поле ввода этого свойства, либо щёлкнуть по кнопке с тремя точками, она выглядит вот так . В результате

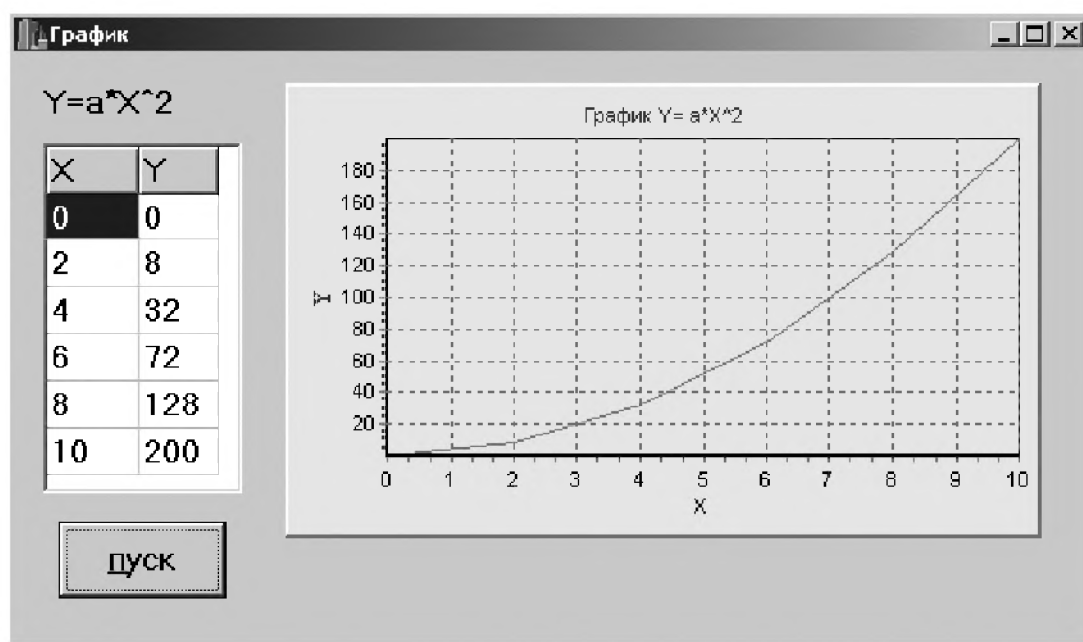


Рис. 5.9. Интерфейс приложения *График*

Имеется и другой путь установки всех параметров свойства *Font* – при помощи набора числовых и строковых констант в полях ввода подсвойств выбранного свойства. Но это очень неудобный путь: упомянутые константы необходимо помнить, знать их допустимый диапазон. Если вы программируете с использованием столь совершенной среды программирования, то используйте все её преимущества. К указанным подсвойствам разумно прибегать лишь при *программном* изменении параметров шрифта. Для реализации такого варианта настройки упомянутые подсвойства удобно использовать в качестве справочной системы: выяснять их значения для параметров, установленных визуально в окне *Выбор шрифта* (рис. 5.10).

Объекты типа *Chart* позволяют увеличивать любой выделенный фрагмент графика (или диаграммы). Испытаем эту возможность в приложении *График*. С этой целью нажмите *левую* кнопку мыши и, не отпуская её, обведите любой выбранный фрагмент графика прямоугольной рамкой, порождаемой с *верхнего левого* угла в *правый нижний* угол. После отпускания кнопки выделенный фрагмент увеличится на *всё* поле окна графика. Рамка *произвольного* размера, порождаемая с *любого другого* угла, *восстанавливает* исходное изображение графика.

Отключение этого режима объекта *Chart1* осуществляется так: в свойстве *UndoZoom* (выключение режима увеличения) в раскрывающемся списке вместо установленной по умолчанию константы *true* выберите константу *false*. Этот же результат достигается, когда в *Редакторе Диаграмм* на странице *Chart* и закладке *General* в разделе *Zoom* убрать птичку в переключателе *AllowZoom*.

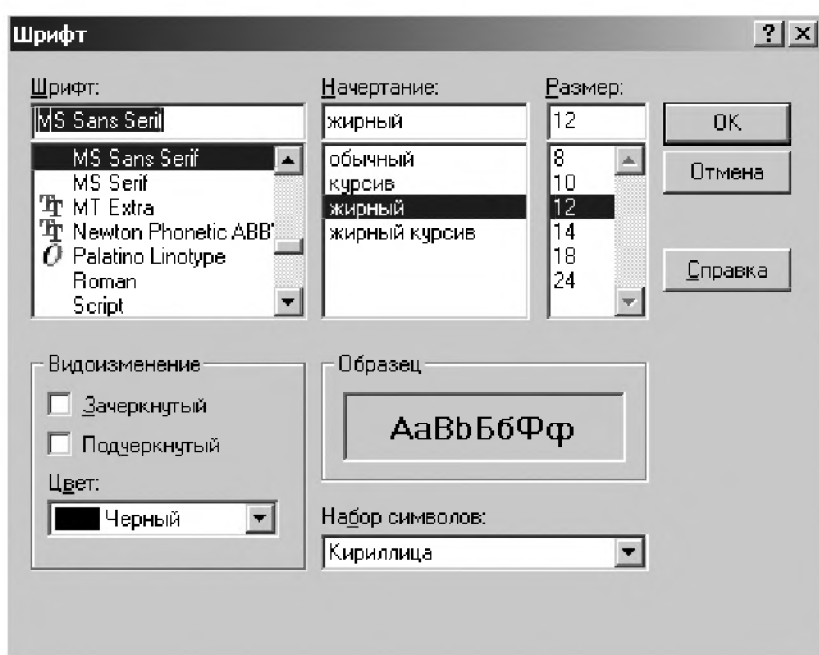


Рис. 5.10. Выбор параметров текста

Можно осуществить ещё один способ возврата к прежнему виду графика: щелчком по окну с графиком при нажатой клавише *Alt*. Опишем последовательность действий по реализации этого способа. Для объекта *Chart1* найдите событие *OnMouseDown* (при нажатии клавиши мыши), дважды щёлкните в его поле ввода. В теле появившейся заготовки функции *Chart1MouseDown*, предназначенной для обработки этого события, наберите следующий код:

```
if (Shift.Contains(ssAlt))
    Chart1->UndoZoom();
```

Здесь функция *UndoZoom()* восстанавливает исходное изображение графика объекта *Chart1*.

Если вместо клавиши *Alt* для возврата к прежнему изображению вы желаете использовать клавишу *Ctrl* или *Shift*, тогда в первой строке кода константу *ssAlt* замените соответственно константой *ssCtrl* или *ssShift*. Все три клавиши будут пригодны для выполнения обсуждаемой операции, если тело функции сделать таким:

```
if (Shift.Contains(ssAlt) || Shift.Contains(ssCtrl) ||
    Shift.Contains(ssShift))
    Chart1->UndoZoom();
```

При увеличенном масштабе графика часто требуется для его более подробного анализа осуществить прокрутку, скроллинг, вдоль осей. Реализация такой возможности осуществляется по умолчанию, вносить изменения в проект не требуется, *Builder* необходимый код включает автоматически. Испытаем прокрутку: указатель мыши расположите на графике, далее нажмите её *правую* клавишу и перемещайте указатель вдоль любой из осей (в произвольном направлении), либо под произвольным углом, если вы желаете осуществить скроллинг по двум осям одновременно.

Такие возможности объекта *Chart1* можно изменить, либо вообще отменить. Для этого следует обратиться к свойству *Allow Panning* (параметры прокрутки, это свойство имеется у графиков и диаграмм, у которых предусмотрены координатные оси) и в его раскрывающемся списке выбрать одну из констант:

- ☐ *pmNone* (***p*anning *m*ode**) – запрет прокрутки;
- ☐ *pmHorizontal* – горизонтальная прокрутка;
- ☐ *pmVertical* – вертикальная прокрутка;
- ☐ *pmBoth* – одновременная прокрутка по горизонтали и вертикали.

Такие настройки можно осуществить и в *Редакторе Диаграмм*. Для этого на странице *Chart* и закладке *General* в разделе *AllowZoom* можно выбрать одну из радиокнопок:

- ☐ *None* – запрет прокрутки;
- ☐ *Horizontal* – горизонтальная прокрутка;
- ☐ *Vertical* – вертикальная прокрутка;
- ☐ *Both* – одновременная прокрутка по горизонтали и вертикали.

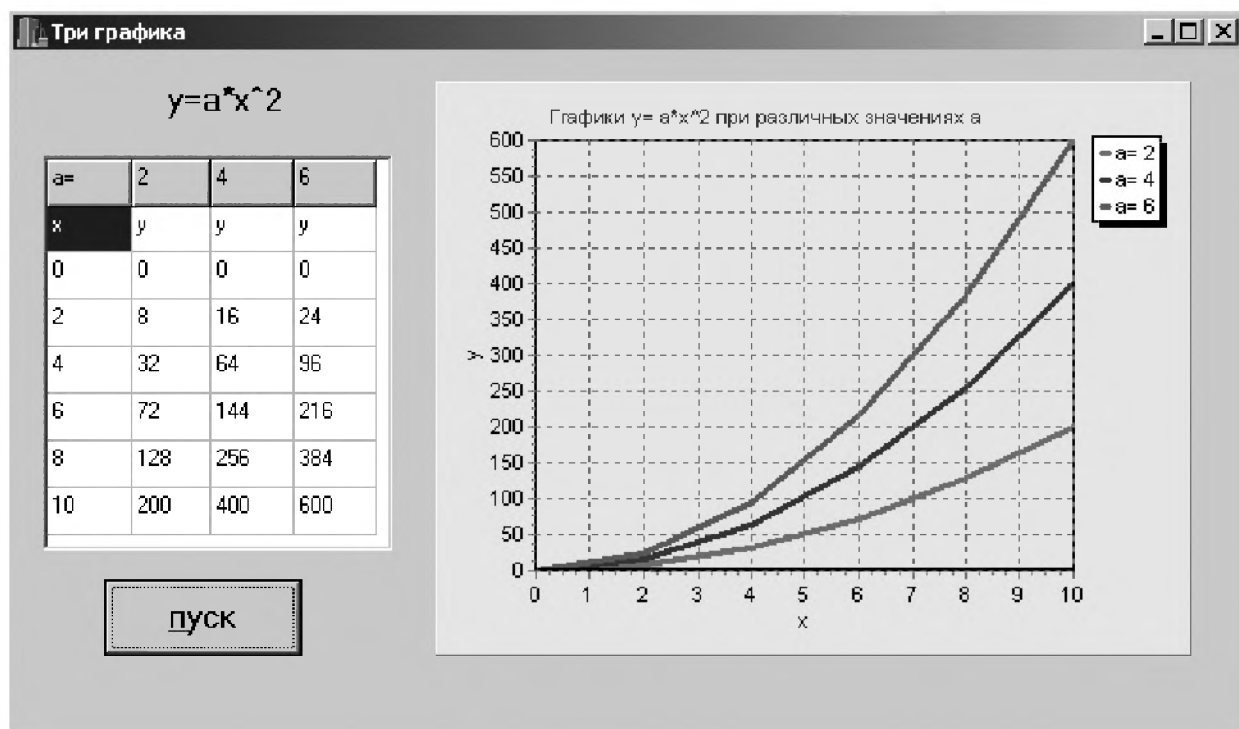
5.2. Построение серии одномерных графиков

Покажем, как осуществляется вывод на экран *серии* из нескольких графиков. Для иллюстрации используем приложение, разработанное на прошлом уроке при изучении вложенных циклов *for*. При этом упростим интерфейс приложения: диапазоны констант, переменных и их приращения пусть вводятся в программе, а не на интерфейсе (по прежним причинам). Интерфейс нового приложения показан на рис. 5.11. Код функции приведен ниже.

```
void __fastcall TTri_Grafika::PyskClick(TObject *Sender)
{
    const int dx= 2, x_Min= 0, n= 6,
              da= 2, a_Min= 2, na= 3;
    //Называем столбцы
    Rezyltat->Cells[0][0]= "a=";
    Rezyltat->Cells[0][1]= "x";
    //Вычисляем ix и iy
    for (int ia, ii= 1; ii<= na; ++ii)
    {
        ia= a_Min + (ii-1)*da;
        Rezyltat->Cells[ii][0]= ia;
        Rezyltat->Cells[ii][1]= "y";
        for (int ij= 2, ix, iy; ij<= n+1; ++ij)
        {
            ix= x_Min + (ij-2)*dx;
            Rezyltat->Cells[0][ij]= ix;
            iy= ia*pow(ix,2);
            Rezyltat->Cells[ii][ij]= iy;
            if (ii== 1)
                Series1->AddXY(ix, iy, "", clRed);
            if (ii== 2)
                Series2->AddXY(ix, iy, "", clBlue);
            if (ii== 3)
                Series3->AddXY(ix, iy, "", clGreen);
        } //for_ij
    } //for_ii
} // PyskClick
```

Построение графиков, соответствующих разным значениям константы *a*, происходит при помощи прежнего объекта *Chart1* в теле внутреннего цикла *for* по переменной *ij*. При этом команда на построение того или иного варианта графика даётся оператором условия *if*. Управляет изменением параметра *ia* внешний цикл *for* по переменной *ii*.

Как же перенастроить объект интерфейса *Chart1*, с тем, чтобы приложение выводило сразу три графика? С этой целью откройте *Редактор Диаграмм* на странице *Series* и закладке *Chart* с окном *Series Title*, показанным на рис 5.1. В разделе 5.1 кнопка *Add* нажималась один раз, после появления окон, приведенных на рис. 5.2 и 5.3, формировали объект *Series1*, имя которого показывалось в окне *Series Title*. Сейчас же эти операции следует повторить три раза. В результате в

Рис. 5.11. Интерфейс программы *Три графика*

окне *Series Title* появятся имена уже трёх объектов с соответствующими обозначениями (см. рис. 5.12).

Кнопка *Delete* (уничтожить) позволяет ликвидировать любой из порождённых объектов, а кнопка *Change* (изменить) даёт возможность выбрать другой шаблон графика для порождённого объекта (при помощи окон, показанных на рис. 5.2 и 5.3).

Предложенные *Builder* имена кривых *Series1*, *Series2*, *Series3*, замените соответственно на $a = 2$, $a = 4$, $a = 6$ (см. рис. 5.11). Это осуществляется при помощи кнопки *Title* (имя кривой) и, появляющегося после её нажатия окна (оно приведено на рис. 5.13). В поле ввода *New Series Title* (новое название кривой) последовательно вводятся упомянутые имена. На рис. 5.13 показан этап процесса переименования последнего графика.

В настоящем приложении на странице *Chart* и закладке *Legend* птичку в окне *Visible* (см. рис. 5.6) следует оставить, поскольку вывод обозначений кривых в данном случае весьма полезен (см. рис. 5.11). В этом же окне при помощи раздела *Position* (положение) можно определить местоположение прямоугольной рамки с обозначениями кривых. С использованием кнопки *Font* выбираются вид, размер, цвет и начертание шрифта. Кнопки *Back Color* (задний план) и *Frame* (рамка) соответственно позволяют выбрать цвет заднего плана литер, цвет и толщину рамки вокруг текста. В разделе *Shadow* (тень) определяется цвет тени от рамки и её толщина.

Задний план графика или цвет панели устанавливается на странице *Chart* при помощи закладки *Panel* (см. рис. 5.14) с применением кнопки *Panel Color*.

Закладка *Axis*, расположенная на этой же странице, уже использовалась в разделе 5.1 (см. рис. 5.7) для ввода имён осей (при помощи вкладки *Title*). На упомянутой закладке имеются также вкладки *Scales* (масштаб основных засечек),

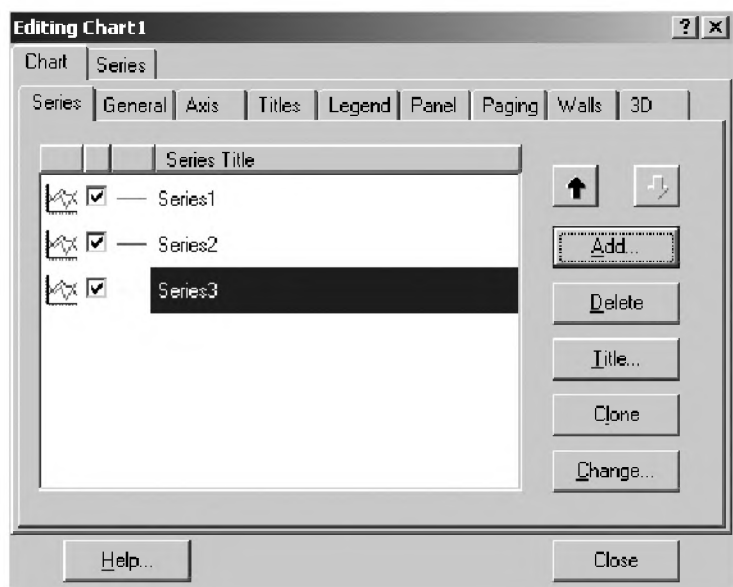


Рис. 5.12. Построение серии из трёх графиков



Рис. 5.13. Переименование кривой графика

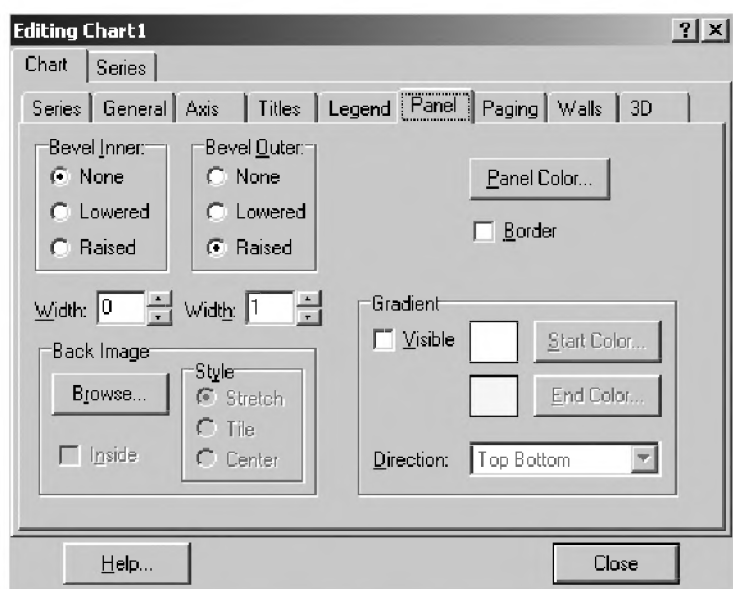


Рис. 5.14. Выбор фона графика

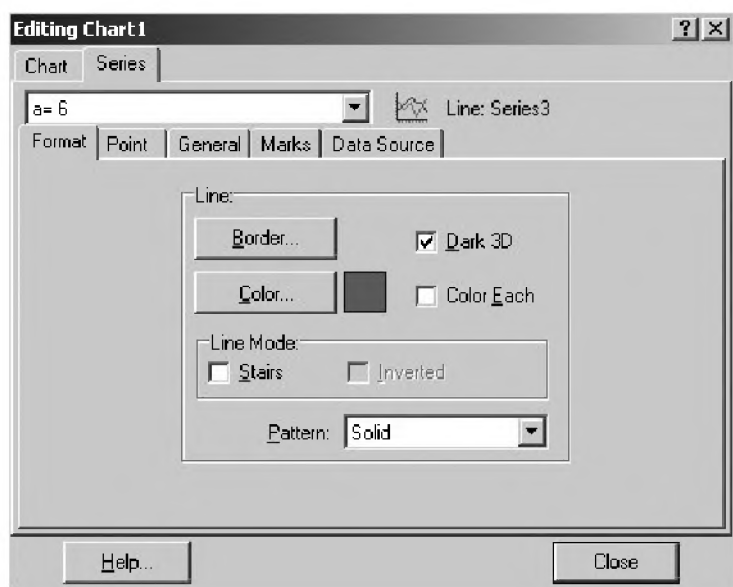


Рис. 5.15. Выбор цвета и толщины линий

Labels (начертание и цвет цифр), *Ticks* (параметры основных засечек и линий сетки), *Minor* (параметры промежуточных засечек и линий сетки), *Position* (местоположение осей), назначение которых ясны из перевода их названий.

Общее название для всех графиков вводится при помощи закладки *Titles*, уже применяемой нами в подразделе 5.1 и показанной на рис. 5.4.

Цвет и толщина кривых графика предлагаются по умолчанию. Изменение этих параметров осуществляется на странице *Series* в закладке *Format*, показанной на рис. 5.15. Следует заметить, что цвет кривой, устанавливаемый при помощи этой закладки, *должен соответствовать* цвету, указанному в функции *AddXY()* (см. листинг). В противном случае оказывается, что цвет обозначений не всегда соответствует выбранным цветам кривых.


Другие закладки этой страниц предназначены для оформления *экспериментальных* точек графика (закладка *Point*); вывода значений ординат над экспериментальными точками графика (*Marks*); определения местоположения координатных осей, разрешения или запрещения вывода на них обозначений (*General*); указания источника данных для выбранной кривой (*Data Source*).

Приведенные в разделах 5.1 и 5.2 учебные графики далеки от совершенства и, видимо, не совсем соответствуют требованиям читателей. Господа, творите самостоятельно, у *Редактора Диаграмм* имеется для этого богатый набор средств.

5.3. Модернизируем интерфейс

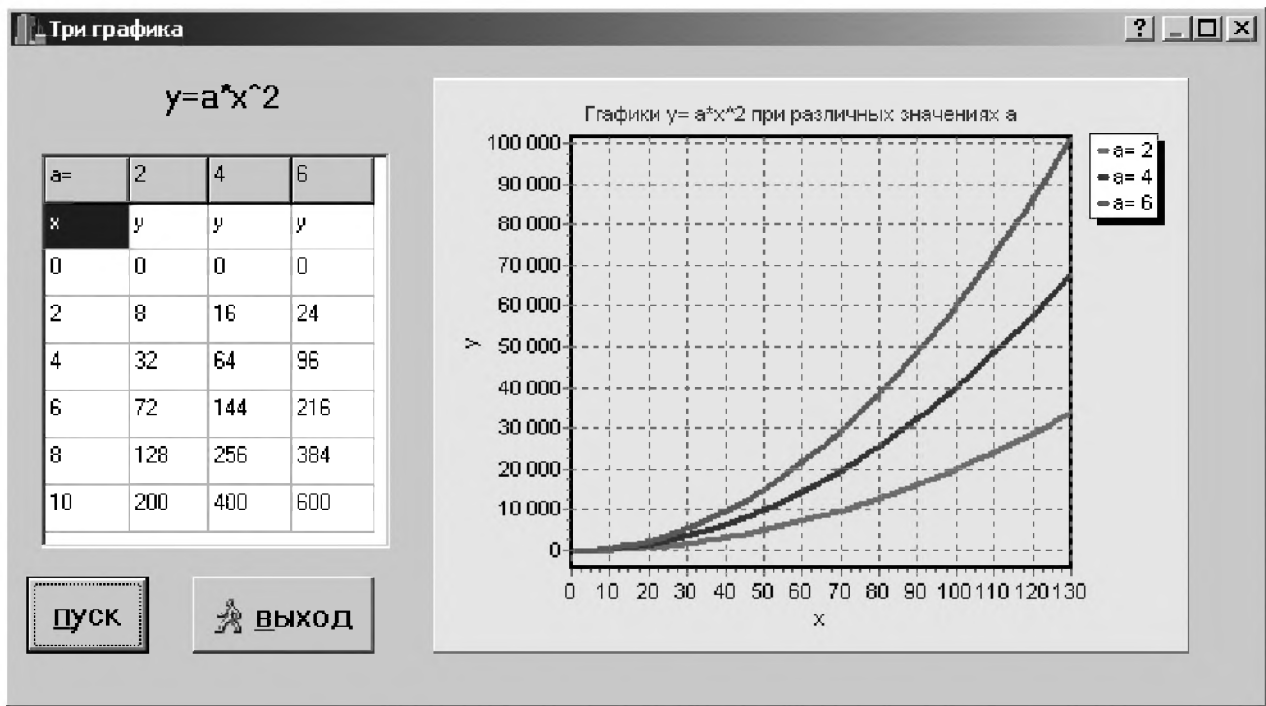
Материал настоящего раздела посвящён повышению привлекательности интерфейса. Этому вопросу уделялось (см. раздел 2.4) и будет уделяться внимание. Привлекательный, оформленный с учётом эргономики, интерфейс существенно способствует повышению эффективности работ, выполняемых при помощи приложения, и его продажам.

5.3.1. Кнопка для выхода из приложения

Пользователи очень любят нажимать кнопки на интерфейсе программ. Поэтому разработчики большинства приложений в угоду таким желаниям снабжают интерфейсы программ кнопкой *Выход*. Вместе с тем завершение *любого* приложения осуществляется стандартным способом: нажатием кнопки .

На рис. 5.16 показан модернизированный интерфейс программы, разработанной в предшествующем разделе (в программе увеличено максимальное значение аргумента). Дополнительная кнопка *Выход* (её имя *Vuxod*) оформлена аналогично кнопке *Пуск*, на ней также подчёркнута буква для ускоренного доступа: выхода из программы при помощи комбинаций клавиш *Alt+в*. Код функции, обрабатывающей нажатие этой кнопки, приведен ниже.

```
void __fastcall TTri_Grafika::VuxodClick(TObject *Sender)
{
    Beep(); //Звуковой сигнал
    ShowMessage("Вы действительно желаете\
```

Рис. 5.16. Модернизированный интерфейс с кнопкой *Выход*

```

завершить работу программы?\n                Зря!");
Beep();
Close();//Выход из приложения
} // VuxodClick

```

Функция *VuxodClick* в своём теле может содержать только одну единственную функцию *Close()*, которая собственно и обеспечивает корректное завершение программы. Однако до начала работы этой функции выводится ещё и предупреждение-шутка, перед появлением которого и после него раздаются короткие звуковые сигналы с использованием функции *Beep()*. Полагаем, у вас не возникнут трудности с внесением упомянутых дополнений в программу.

В строковой константе функции *ShowMessage* используется обратный слеш (обратная косая черта), набранный без пробела со *строчной* латинской буквой *n*. Такие два символа осуществляют *перенос* текста (расположенного после них) на соседнюю строку выводимого окна. Обратный же слеш без какой-либо литеры (и последующих пробелов) применяется при переносе текста в программе (внутри всплывающего окна перенос текста в этом месте не произойдёт). В последнем случае слеш используется только для структурирования текста программ. Окно с предупреждением-шуткой показано на рис 5.17.

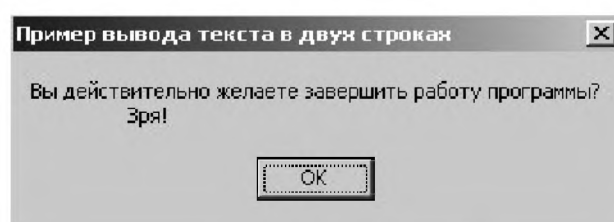


Рис. 5.17. Окно с предупреждающим текстом


Для выхода из программы вместо функции *Close()* можно использовать функцию *exit(0)* с нулевым значением аргумента, а также функцию *Terminate()* невидимого объекта *Application*, вызов которой выглядит вот так:

```
Application->Terminate();//Выход из приложения
```

Применение всех этих способов завершения программы не требуют от разработчика приложения подключения каких-либо библиотек (заголовочных файлов). Всё, что необходимо для их использования *Builder* к программе уже подключил. Испытайте эти варианты завершения работы приложения. Если у вас всё получилось, тогда займёмся дополнительным развлечением пользователя перед окончанием работы с программой, ведь должен же он немного отдохнуть от вашего продукта.

5.3.2. Показ стандартных мультфильмов

Давайте сделаем так, чтобы перед выходом из программы демонстрировался мультфильм, например, такой, как при копировании, уничтожении или поиске файлов. В нашем приложении мультипликация используется *лишь* для испытания механизма её реализации. Истинное назначение анимации состоит в том, чтобы информировать пользователя о сущности выполняемого длительного процесса (около или более нескольких секунд). Ниже перечислим необходимые действия для создания анимации.

На вкладке *Палитры Компонентов* с именем *Win32* выделите такой компонент . Затем щёлкните мышью по свободной части формы (например, в её левом верхнем углу), штриховая рамка, предназначенная для размещения нового объекта (типа *TAnimate*) интерфейса, частично выйдет за границы формы. Не волнуйтесь, не старайтесь исправить ситуацию: при работе программы объект не будет таким большим. В *Инспекторе Объектов* можно изменить имя нового объекта, однако оставьте его в варианте по умолчанию: *Animate1* (анимация 1).

Далее активизируйте свойство *CommonAVI* (стандартные файлы AVI). В раскрывающемся списке этого свойства имеется 8 имён файлов с анимационными картинками (или кадрами), размеры картинок в каждой серии одинаковы. Быстрая смена таких картинок создаёт иллюзию движения предмета, поскольку предметы изображены на картинках в разных фазах движения. Просмотрите некоторые из этих файлов. Для этого после выбора очередного имени файла его свойству *Active* вместо исходного значения *false* установите значение *true*, при этом запустится выбранная мультипликация (она непрерывно повторится).

Остановите свой выбор, например, на файле *aviFindComputer* (поиск компьютера). После выбора файла границы объекта автоматически уменьшатся до размеров его картинок (кадров). Это происходит вследствие того, что по умолчанию в свойстве *AutoSize* установлено значение *true*. Не изменяйте его, размер кадров в любом случае увеличить не удастся.

Теперь верните свойству *Active* значение *false* (демонстрация выбранного мультфильма прекратится), свойству *Visible* объекта *Animate1* установите также значение *false* (с тем, чтобы первая картинка мультфильма не была видна до нажатия кнопки *Выход*) и перейдите в *Редактор Кода*, где в функции *VuxodClick* добавьте несколько строк, после чего она будет выглядеть вот так:

```
void __fastcall TTri_Grafika::VuxodClick(TObject *Sender)
{
    Beep(); //Звуковой сигнал
    Animate1->Visible= true;
    Animate1->Play(1, 23, 3);
    ShowMessage("Вы действительно желаете\
завершить работу программы?\n\n          Зря!");
    Beep();
    Close();
} //VuxodClick
```

В этой функции строка

```
Animate1->Visible= true;
```

приводит к тому, что ранее невидимый объект *Animate1* становится видимым на интерфейсе, поскольку его свойству *Visible* присваивается значение *true* (вместо исходного *false*). В строке

```
Animate1->Play(1, 23, 3);
```

функция *Play* в объекте *Animate1* запускает картинки выбранного нами файла с первой по двадцать третью (последнюю) и такую демонстрацию повторяет три раза. Таким образом, первый параметр функции *Play* – это номер кадра, с которого начинается демонстрация, второй параметр указывает, каким кадром демонстрация заканчивается. Третий параметр задаёт число повторений мультфильма. Если он равен нулю, то мультфильм повторяется бесконечно.

Как поступить в том случае, когда вы не знаете, сколько кадров содержится в фильме? Имеется два способа:

- ☐ Посмотреть номер последнего кадра в свойстве *StopFrame*.
- ☐ Поставить на место этого параметра число, заведомо большее возможного количества кадров мультфильма, например, 10 000.

Если в функции *VuxodClick* не использовать функцию *ShowMessage()*, то мультфильм демонстрироваться не будет: функция *Close()* не дожидаясь конца демонстрации мультфильма, вступит в свои права. Для показа мультфильма следует в функции *VuxodClick* исключить также и функцию *Close()*. Выход из приложения после демонстрации *последнего* кадра мультфильма осуществите теперь при помощи события (для объекта *Animate1*) *OnStop* и функции по его обработке *Animate1Stop()*:

```
void __fastcall TTri_Grafika::Animate1Stop(TObject *Sender)
{
    Close();
} //Animate1Stop
```

5.3.3. Стандартные кнопки Windows

При оформлении кнопок, выполняющих какие-то типовые действия (выход из программы, положительный или отрицательный ответы, отмена и др.), можно использовать *стандартные* кнопки (см. рис. 5.18), на которых помимо ясных заголовков изображены ещё и картинки-подсказки упомянутых действий.

Такие кнопки являются объектами типа *TBitBtn* (*Bit Button* – кнопка с битовой картинкой), их заготовка расположена на вкладке *Additional* *Панели Компонентов*. Свойство *Kind* (вид) объекта этого типа позволяет выбрать одну из кнопок, показанных на рис. 5.18. Заголовки кнопок изменяются при помощи свойства *Caption*. При этом в отличие от объектов типа *TButton*, имеется возможность устанавливать *цвет* шрифта. Обращаем внимание на то, что в кнопке с заголовком *Close* выход из приложения осуществляется автоматически, без явного применения функции *Close()*.

Используйте одну из этих кнопок для нашего приложения. С этой целью расположите кнопку такого типа на интерфейсе, выполните двойной щелчок по ней, а затем в заготовку сгенерированной функции скопируйте тело функции *VuxodClick*. После этого прежнюю кнопку *Выход* (но не её функцию) уничтожьте, а её заголовок и имя присвойте соответственно заголовку и имени новой кнопки.

Теперь единственная кнопка приложения выглядит так : .

При её нажатии будет осуществляться выход из программы с предупредительными сигналами, текстом и мультфильмом.

5.3.4. Размещение картинок на кнопках

На объектах типа *TBitBtn* можно размещать изображения в формате *bmp*. При этом габаритные размеры картинки *не должны* превышать габариты кнопки. Нарисуйте, например, картинку-надпись в графическом редакторе *MS Paint* и сохраните её в файле с расширением *bmp* в каталоге нового проекта *Privet*. Далее на форме этого приложения разместите лишь кнопку типа *TBitBtn*, в её свойстве *Caption* сотрите заголовок по умолчанию *BitBtn1*. Теперь для этой кнопки в её свойстве *Glyph* (рельефно выделенный знак или символ) при помощи кнопки с тремя точками (или двойным щелчком в поле ввода свойства) вызовете редактор

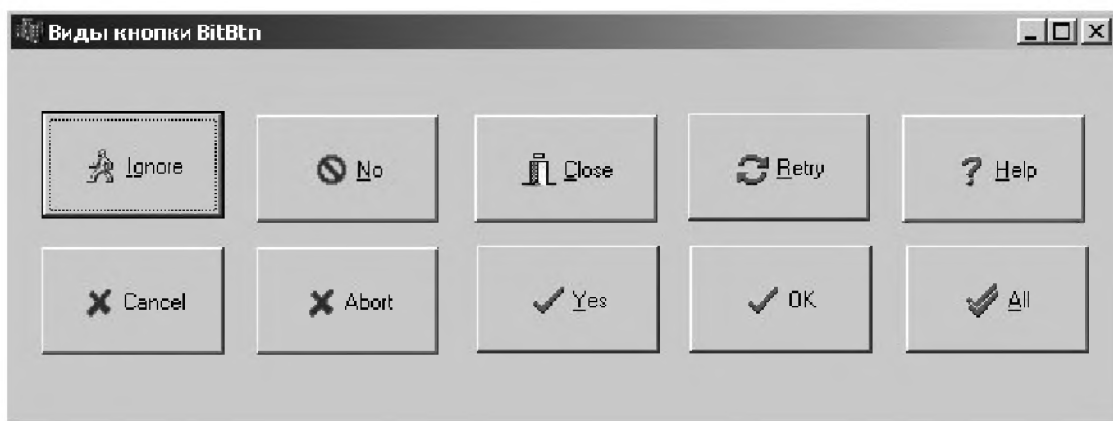


Рис. 5.18. Вид стандартных кнопок

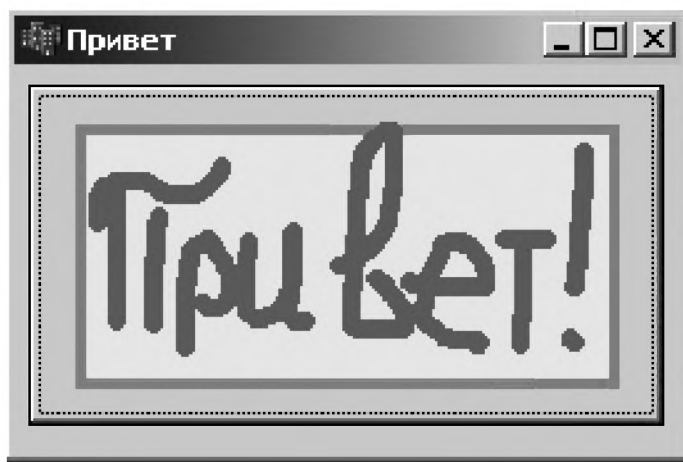



Рис. 5.19. Размещение картинки-надписи на командной кнопке типа *TBitBtn*

картинок (*Picture Editor*). С использованием кнопки *Load* (загрузить) этого редактора, кнопки *Открыть* появившегося окна *Load Picture* (загрузить картинку) и кнопки *ОК* окна *Picture Editor* картинку расположите (загрузите) на кнопку. На рис. 5.19 показан интерфейс приложения с заголовком *Привет*, на кнопке которого расположена картинка-надпись.

5.3.5. Размер формы и граничные пиктограммы

Свойство формы *BorderStyle* (виды границ) позволяет разрешить или запретить изменение габаритов интерфейса программы. При этом можно выводить *все три* стандартные пиктограммы заголовка интерфейса  (далее граничные пиктограммы), либо *только последнюю* из них.

По умолчанию в этом свойстве устанавливается значение *bsSizeable* (*bs* – от *BorderStyle*), при котором нет блокировки изменения габаритов формы, и имеются все три граничные пиктограммы.

При значении *bsSingle* (*single* – единственный) габариты формы неизменны, имеются все три граничные пиктограммы.

При значениях *bsDialog* и *bsToo Window* также изменение габаритов блокируется, однако имеется только последняя граничная пиктограмма.

Значение *bsSizeToo Win* позволяет вывести только последнюю пиктограмму и изменять габариты интерфейса.

Значение *bsNone* осуществляет блокировку изменения габаритов интерфейса и при нём не выводится *ни одна* граничная пиктограмма.

Габариты формы наиболее удобно устанавливать при помощи указателя мыши (методом протягивания). Однако для этих целей применимы также свойства *ClientHeight* (установка высоты) и *ClientWidth* (установка ширины), в полях ввода которых набираются конкретные значения габаритов (измеряются в пикселях). Указанные свойства служат и для *программного* способа установки размеров интерфейса.

Имеется также *отдельный* редактор для граничных пиктограмм окна: это свойство *BoderIcon*. У него имеется четыре подсвойства, каждое из которых можно сделать активным (*true*), либо пассивным (*false*).

При пассивности подсвойства *biSystemMenu* ни одна граничная пиктограмма не выводится. Для реализации остальных подсвойств подсвойство *biSystemMenu* следует сделать активным.

В этом случае при активности подсвойства *biMaximize* и пассивности подсвойства *biMinimize* оказывается пассивной (но выводится) первая пиктограмма (она называется пиктограммой минимизации, предназначена для сворачивания окна на панель задач), две последние пиктограммы выводятся и являются активными.

Если же пассивно подсвойство *biMaximize* и активно подсвойство *biMinimize*, то тогда оказывается пассивной (но выводится) вторая пиктограмма (максимизации, переключатель между полноэкранным и нормальным окнами), выводятся и активны первая и последняя граничные пиктограммы.

Установка в подсвойствах *biMinimize* и *biMaximize* константы *false* приводит к тому, что кнопки минимизации и максимизации не выводятся, активна и выводится только последняя пиктограмма.

Далее сделайте активными все предшествующие подсвойства и для подсвойства *biHelp* также выберите значение *true* (по умолчанию в нём находится *false*). Теперь при наведении указателем мыши на любую из пиктограмм появляется всплывающая подсказка, соответственно: *Свернуть*, *Развернуть* и *Заккрыть*. Если же при активности последнего подсвойства одну из пиктограмм сделать неактивной, то при наведении указателя мыши на такую пиктограмму подсказка появляться не будет.

Вопросы для самоконтроля

- ☐ С какой целью в программах используются знаки «\» и «\n»?
- ☐ Как узнать число кадров в стандартных мультфильмах объекта типа *TAnimate*, можно ли это число заменить в функции *Play()* другим числом?
- ☐ Назовите свойства, используемые для редактирования граничных пиктограмм, блокировки и разрешения изменения габаритных размеров окна программы. В чём заключается различие этих свойств?

5.4. Задачи для программирования

Задача 5.1. Модернизируйте приложение *Три графика* таким образом, чтобы последний кадр анимации исчезал с интерфейса после завершения демонстрации мультфильма (до нажатия кнопки *ОК* в появившемся окне с текстом предупреждения).

Задача 5.2. Сделайте так, чтобы на интерфейсе приложения *Три графика* все результаты появлялись бы сразу же после запуска программы и отсутствовала кнопка *Пуск*. При этом вместо трёх операторов *if* примените оператор множественного выбора *switch*.

5.5. Варианты решений задач

Для решения задачи 5.1 выделите объект *Animate1* интерфейса приложения *Три графика*, перейдите в *Инспектор Объектов* на вкладку *События* и там выберите событие *OnStop* (по завершению демонстрации), сделайте двойной щелчок в поле его ввода и в заготовке функции, появившуюся в *Редакторе Кода*, впишите операторы *Animate1->Visible=false*; В законченном виде функция выглядит так:

```
void __fastcall TTri_Grafika::Animate1Stop(TObject *Sender)
{
    Animate1->Visible= false;
} //Animate1Stop
```

Решение задачи 5.2 лучше осуществить в два этапа. Действия первого этапа состоят в следующем. Выделите форму приложения *Три графика*, перейдите в *Инспектор Объектов* на вкладку *События* и там выберите событие *OnCreate*, сделать двойной щелчок в поле его ввода. Далее, в заготовку функции, появившуюся в *Редакторе Кода*, скопировать тело функции *PyskClick*. Затем испытайте программу. При её запуске сразу появится результат, показанный на рис. 5.16.

Второй этап работы: тело функции *FormCreate* измените на случай применения оператора *switch*. Ниже показана функция *FormCreate* в законченном виде.

```
void __fastcall TTri_Grafika::FormCreate(TObject *Sender)
{
    const int dx= 2, x_Min= 0, n= 66,
              da= 2, a_Min= 2, na= 3;
    //Называем столбцы
    Rezyltat->Cells[0][0]= "a=";
    Rezyltat->Cells[0][1]= "x";
    //Вычисляем ix и iy
    for (int ia= a_Min, ii= 1; ii<= na; ++ii)
    {
        ia= a_Min + (ii-1)*da;
        Rezyltat->Cells[ii][0]= ia;
        Rezyltat->Cells[ii][1]= "y";
        for (int ij= 2, ix, iy; ij<= n+1; ++ij)
        {
            ix= x_Min + (ij-2)*dx;
            Rezyltat->Cells[0][ij]= ix;
            iy= ia*pow(ix,2);
            Rezyltat->Cells[ii][ij]= iy;
            switch (ii)
            {
                case 1: Series1->AddXY(ix, iy, "", clRed);
                        break;
                case 2: Series2->AddXY(ix, iy, "", clBlue);
                        break;
                case 3: Series3-> AddXY(ix, iy, "", clGreen);
            } // switch
        } //for_ij
    } //for_ii
} //FormCreate
```

Теперь кнопку *Пуск* можно уничтожить, поскольку она больше не нужна. Если вы уничтожите и её функцию *PyskClick*, то при запуске программы появится сообщение об ошибке на этапе линковки. Дело в том, что в классе формы эта функция была объявлена, а её реализации больше нет. Поэтому необходимо уничтожить и объявление. Для этого перейдите в *Редактор Кода*, а затем нажмите сочетание клавиш *Ctrl+F6*. В результате вы перенесётесь в класс формы, найдите там и уничтожьте следующую строку: `void __fastcall PyskClick(TObject *Sender);`. После этого *Builder* успокоится, и программа запустится. Все эти рекомендации более подробно разъясняются на следующем уроке.

Урок 6. Одномерные массивы

6.1. Зачем нужны массивы?

Рассмотрим практическую задачу, после её решения вам станет ясно, что собой представляют массивы и зачем они нужны.

Имеется ведомость успеваемости группы учащихся по информатике. Оценки первых десяти учащихся группы представлены в таблице.

Таблица 6.1. Ведомость успеваемости учащихся по информатике

1	2	3	4	5	6	7	8	9	10
5	3	12	9	8	6	10	11	7	8

Требуется определить среднюю и минимальную оценки, а также число учащихся, у которых оценка выше 7.

6.1.1. Интерфейс приложения

Интерфейс приложения после ввода исходных данных и нажатия кнопки *Пуск* показан на рис. 6.1.

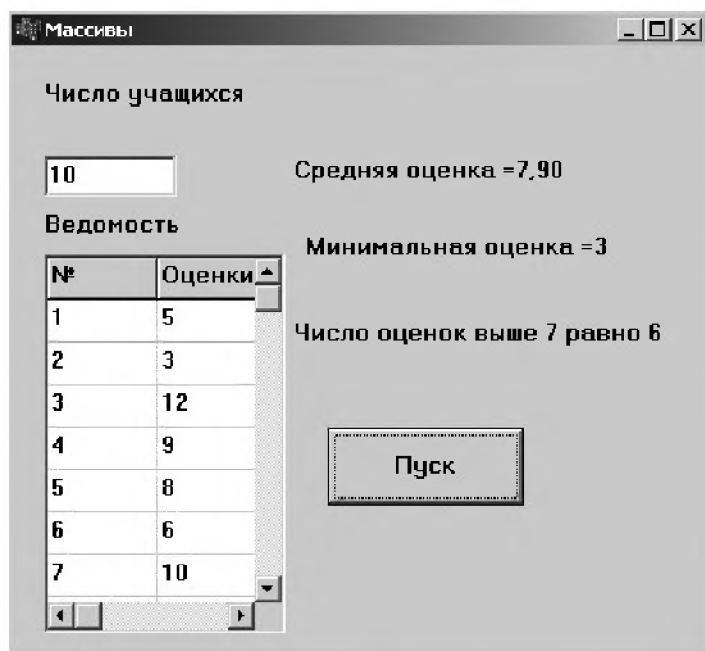


Рис. 6.1. Интерфейс приложения *Массивы*

Поле ввода, в которое заносится число учащихся, называется *Ch_Ych*, таблице с оценками присвоено имя *Tab*, а командной кнопке – *Pysk*. Поля вывода результатов имеют имена: *Sredn_Ocenka* (средняя оценка), *Min_Ocenka* (минимальная оценка), *Bolshe_7* (число учащихся, у которых оценка больше 7). Интерфейс с такими объектами вы легко можете собрать самостоятельно. Однако после запуска приложения осуществить ввод каких-либо значений в таблицу строк вам не удастся: по умолчанию ввод данных в ячейки объектов типа *TStringGrid* запрещён.

Для разрешения такого ввода в свойстве *Options* (параметры настройки) таблицы строк необходимо выбрать подсвойство *goEditing* (переход к редактированию) и в раскрывающемся списке вместо установленного по умолчанию значения *false* выбрать *true*. Теперь содержимое таблицы можно редактировать: вводить в её ячейки различные значения, изменять их либо стирать. Ввод нового значения заменяет предыдущее.

Свойство *Options* состоит из ряда подсвойств. Перечислим назначения некоторых из них: *goFixedVertLine*, *goFixedHorzLine* и *goVertLine*, *goHorzLine* определяют наличие разделительных вертикальных и горизонтальных линий соответственно в фиксированных и нефиксированных ячейках; *goRowSizing* и *goColSizing* дают возможность изменять с помощью мыши высоту строк и ширину столбцов; *goColMoving* и *goRowMoving* разрешают перемещать мышью столбцы и строки.

Напоминаем, выбор числа строк и столбцов (с учётом нулевых, предназначенных для вывода заголовков) таблицы осуществляется свойствами *RowCount* и *ColCount*. Свойства *FixedCol* и *FixedRow* определяют число фиксированных, не прокручиваемых столбцов и строк. Цвет фона ячеек-заголовков устанавливается свойством *FixedColor*. При помощи свойств *LeftCol* и *TopRow* выбираются соответственно индексы первого видимого на экране в данный момент прокручиваемого столбца и первой видимой прокручиваемой строки.

Большинство упомянутых свойств и подсвойств в нашей задаче имеют те значения, которые установлены в *Builder* по умолчанию. Их назначение указаны для иллюстрации возможностей объекта типа *TStringGrid*. Ещё раз обращаем внимание на то, что в клетках таблицы может размещаться *только текстовая* информация, ведь не случайно этот объект называется *таблицей строк*. Поэтому для того, чтобы значения, введенные в её ячейках, скопировать (записать, присвоить, запомнить) в числовые переменные необходимо преобразование из текстового формата в числовой (*int*, *float* или *double*). Указанное предупреждение для свойства *Cells* объектов типа *TStringGrid* относится также и к свойству *Text* объектов типа *TEdit*.

Интерфейс программы при её запуске определяет функция *FormCreate*, код её тела приведен ниже.

```
Ch_Ych->Text= "10";
Tab->ColCount= 2;
Tab->RowCount= 11;
Tab->Cells[0][0]= "№";
Tab->Cells[1][0]= "Оценки";
```


можно разместить *только* по *одному* числовому значению. Переменные, в которых можно сохранять лишь по одному значению, называются простыми.

Как же запомнить целую *вереницу* чисел таблицы? В принципе, каждое число, расположенное в клетке таблицы, можно сохранить в отдельной переменной. Однако применять целый ряд переменных для хранения *всех* чисел таблицы весьма неудобно, даже если в таблице имеется только десять ячеек.

Поэтому во всех языках программирования высокого уровня для хранения *совокупности* однотипных данных выделяется *специальная* переменная, называемая *массивом*. Массив относится к структурированному типу данных и состоит из *целой последовательности* ячеек памяти одинакового объёма. Вся группа таких ячеек не разбросана по различным областям сегмента данных, а находится в *едином* блоке, в котором соседние ячейки массива занимают *смежные* ячейки памяти. Во всех ячейках массива можно хранить данные только *одного* заданного типа.

Номер элемента массива ещё именуют индексом, а сам массив – переменной с индексами. Итак, массив – это последовательность *одинаковых* по объёму ячеек памяти с единым именем. Объём ячейки определяется типом данных, которые в ней предполагается хранить. Количество ячеек *заранее* устанавливается при описании обсуждаемой составной или *структурированной* переменной.

Как уже отмечалось, у каждой ячейки в массиве имеется свой номер, также как, например, у вагонов поезда. Вот только первый вагон поезда имеет номер «1», а первой ячейке массива *всегда* присваивается номер «0». В C++ нумерация элементов массива, начинается с нуля, а не с единицы. Такое правило заимствовано из языка C (а родители C скопировали такой порядок из языка *Assembler*).

Тридцать лет назад, когда царствовал прародитель C++ *Builder* и C++, упомянутый порядок нумерации позволял *несколько* повысить быстродействие программ. Для современных компьютеров такое повышение быстродействия совершенно неактуально. Однако приходится учитывать ранее введенное правило: первая ячейка массива имеет номер «0», поэтому у последней ячейки массива из n элементов номер равен $(n - 1)$.

Число элементов (ещё говорят размер, длина) и тип *статических* массивов задаются при объявлении переменных-массивов, они *не могут* изменяться в ходе работы программы. Статическими называются массивы, память для размещения которых, выделяется ещё *до* начала работы программы, в ходе её трансляции.

В нашей учебной программе массив оценок обозначим именем *iOcenka*, а для *максимально возможного* числа его элементов, в целях общности, использована глобальная константа n (см. выше). Объявление локального массива *iOcenka* в функции *PyskClick* выглядит так:

```
int iOcenka[n]; //Место для хранения n оценок
```

Здесь после имени *iOcenka* в *квадратных скобках* указана длина массива (число его элементов, ячеек, компонент). Тип элементов (в нашем примере это тип *int*) как и для простых переменных приводится *перед* идентификатором переменной с индексом.

Builder при анализе текста программы находит идентификатор массива (по наличию квадратных скобок после его имени), после чего выделяет требуемое количество ячеек памяти заданного размера. В нашем примере для массива *iOcenka* выделяется 40 байт (10 ячеек·4 байта = 40 байт). Во всех *числовых* ячейках *локального* массива при его порождении (также как и при порождении локальных числовых переменных) находятся неприсвоенные значения (случайные числа).

Как уже отмечалось, под массив выделяется сплошной блок ячеек оперативной памяти, который на момент этой операции является незадействованным под данные. Однако до этого момента упомянутые ячейки могли полностью либо частично использоваться. В этом случае в них находятся совокупности каких-то двоичных цифр, которые могут интерпретироваться по-разному в зависимости от того какой тип ячеек у вашего массива. Только строковой переменной типа *String* (см. урок 12) при её порождении автоматически присваивается пустая строка.

Заполняются ячейки массива при помощи оператора присваивания, после имени массива в *квадратных скобках* указывается номер той ячейки, в которую требуется поместить нужное число (или текст, литеру). Вот так в ячейку с номером *ij* массива *iOcenka* вводится, например, число 2002:

```
iOcenka[ij] = 2002;
```

Покажем теперь фрагмент кода функции *PyskClick*, который обеспечивает наполнение элементов массива *iOcenka* данными, взятыми из ячеек объекта *Tab*:

```
//Заносим исходные данные в массив iOcenka[n]
for (int ij= 0; ij< m; ++ij)
    iOcenka[ij]= StrToInt(Tab->Cells[1][ij + 1]);
```

Вопрос для текущего самоконтроля

- ☐ Почему в вышеприведенном цикле *for* индекс последнего шага ($m - 1$) не равен m – числу элементов массива *iOcenka*?

6.1.3. Суммирование элементов массива

Для вычисления средней оценки выбранной группы учащихся найдём вначале сумму всех оценок: просуммируем все элементы массива с оценками. Вот как это делает фрагмент разрабатываемой нами функции:

```
//Суммируем оценки
fSredn_Ocenka= 0; // Очистка ячейки
for (int ij= 0; ij< m; ++ij)
    fSredn_Ocenka += iOcenka[ij];
```

Вначале локальная переменная *fSredn_Ocenka*, предназначенная для хранения средней оценки, используется не по своему основному назначению. Прежде чем записать в неё среднюю оценку, применим её для *суммирования* всех оценок. Такое совмещение сфер деятельности сокращает число используемых переменных, объём памяти, выделяемый под переменные.

При помощи первого оператора присваивания в *fSredn_Ocenka* записано ну-

левое значение. Этим действием осуществлена операция предварительной *очистки* переменной. Без такой операции случайное исходное значение переменной в зависимости от его знака приведёт к занижению либо к завышению конечной суммы.

Если сделать переменную *fSredn_Ocenka* глобальной (объявить, например, в файле реализации), то тогда в функции *PyskClick* её можно не очищать (не обнулять), ведь после порождении *всех* глобальных числовых переменных в них и без такой операции записывается нуль. Однако согласно правилам хорошего стиля программирования настоятельно рекомендуется операцией очистки *никогда* не пренебрегать.

Это необходимо для разработки таких программ, которые легко допускают «безболезненную» модернизацию. Ведь при будущем возможном совершенствовании программы впереди блока суммирования эта переменная может применяться для хранения каких-то отличных от нуля значений. Блок суммирования должен быть *независим* от внешних условий, в этом случае его всегда можно без изменений применить в *любом месте программы*.

Цикл *for* на первом своём шаге к исходному нулевому значению *fSredn_Ocenka* добавляет первую оценку из нулевого элемента массива *iOcenka[0]*. Поэтому после завершения этого шага в *fSredn_Ocenka* находится оценка первого учащегося.

Второй оборот цикла приведёт к сложению *fSredn_Ocenka* с оценкой из первой ячейки *iOcenka[1]*. Поскольку перед таким сложением в *fSredn_Ocenka* размещена только оценка из нулевой ячейки массива, то в результате второго шага в *fSredn_Ocenka* записывается сумма первых двух элементов массива *iOcenka* (нулевого и первого).

Третий и последующие циклы последовательно добавляют к *fSredn_Ocenka* оценки из соответствующих компонент массива. Последний шаг цикла завершит суммирование элементов массива *iOcenka* добавлением к сумме из $(n - 1)$ слагаемых последнего *n*-го слагаемого (в нашем случае девятого элемента).

6.1.4. Определение экстремума

Найдём теперь минимальную оценку в указанном массиве-ведомости. С этой целью положим вначале, что минимальный элемент *iMin_Ocenka* находится в нулевой ячейке *iOcenka[0]*. Раз так, то скопируем его из указанной ячейки в переменную, специально предназначенную для хранения такой величины:

```
iMin_Ocenka= iOcenka[0];
```

Однако использованное предположение может не соответствовать действительности. Как выяснить истину? Для проверки предварительного решения сравним *iMin_Ocenka* с оценкой, находящейся в ячейке *iOcenka[1]*. Если окажется, что эта оценка не меньше *iMin_Ocenka*, тогда продолжим сравнение *iMin_Ocenka* с оценками из последующих ячеек массива. Когда в одной из компонент массива, например в *iOcenka[ij]*, окажется значение, меньшее, чем в

iMin_Ocenka, то это значение присвоим *iMin_Ocenka* и продолжим далее сравнивать новое значение *iMin_Ocenka* с элементами *iOcenka[ij+1]*, *iOcenka[ij+2]* и так далее.

При таком сравнении может и не найдётся больше ни одного элемента, который бы претендовал бы на минимальное значение. Если же такой претендент всё же отыщется, тогда методом присваивания поручить и ему роль «дежурного» минимального значения. Эстафетная палочка *iMin_Ocenka* будет *последовательно* передаваться из рук в руки претендентов на всей «дистанции» массива. В результате, как только закончится сравнение *iMin_Ocenka* с *iOcenka[n - 1]*, в *iMin_Ocenka* *обязательно* окажется минимальный элемент массива *iOcenka*.

Словесное описание алгоритма поиска *iMin_Ocenka* реализуется следующим фрагментом программы:

```
//Определяем минимальную оценку
iMin_Ocenka= iOcenka[0];
for (int ij= 1; ij< m; ++ij)
    if (iOcenka[ij]< iMin_Ocenka)
        iMin_Ocenka= iOcenka[ij];
```

Вопрос для текущего самоконтроля

- ☐ Будет ли правильно работать этот фрагмент программы, если *ij* изменять не от 1 до $m - 1$, а от 0 до $m - 1$?

Можно стартовое минимальное значения положить равным *iOcenka[ir]*, где *ir* – номер любой ячейки от 1 до $n - 1$. В этом случае поиск минимального значения следует начинать от *ij= 0*. Такой поиск также будет успешным, вот только число его шагов на один шаг увеличится, поэтому его выполнение затянется на более длительное время.

В данной программе это ничтожное мгновение. Однако сравнения могут выполняться по очень сложным алгоритмам, требующим для своей работы большого времени. И вот тогда экономия окажется ощутимой и ваша рачительность не пропадёт зря. Приучайтесь сокращать время работы программы и на простых операциях.

На практике кроме поиска минимального элемента массива приходится находить и другой его экстремум – максимум. Для решения такой задачи можно применить прежний блок программы, в нём лишь знак « $<$ » изменить на « $>$ », а идентификатор *iMin_Ocenka* – заменить на *iMax_Ocenka*.

6.1.5. Счетчик

Подсчитаем число учащихся *iBolshe_7*, у которых оценка больше 7. С этой целью просмотрим каждый элемент массива *iOcenka*, для перехода от одного элемента массива к другому применим цикл *for*:

```
//Определяем число учащихся, у которых оценка больше 7
iBolshe_7= 0; // Очистка ячейки
for (int ij= 0; ij< m; ++ij)
    if (iOcenka[ij]> 7)
        iBolshe_7++;
```

В этом фрагменте также выполняется суммирование, поэтому имеется очистка переменной *iBolshe_7*. Условие, положенное в основу отбора суммируемых ячеек, реализуется использованием сокращённой формы оператора *if*.

Выше приведен алгоритм определения числа элементов массива со значениями, превышающими 7. Аналогичным путём можно также узнать:

- ☐ число отрицательных или положительных элементов массива;
- ☐ число элементов, равных заданному числу;
- ☐ число элементов меньших или больших заданного числа;
- ☐ число элементов, находящихся в интервале заданных значений.

Для решения таких задач применяется алгоритм, называемый *счетчиком*.

6.1.6. Функция PyskClick в законченном виде

Выше подробно рассмотрены основные этапы решения учебного задания. Просмотрите теперь всё тело функции *PyskClick*.

```
const int m= StrToInt(Ch_Ych->Text); //Текущее число учащихся
float fSredn_Ocenka; //Средняя оценка
int iMin_Ocenka, //Минимальная оценка
iBolshe_7; //Число учащихся, у которых оценка больше 7
int iOcenka[n]; //Массив оценок

//Заносим исходные данные
for (int ij= 0; ij< m; ++ij)
    iOcenka[ij]= StrToInt(Tab->Cells[1][ij+1]);

//Суммируем оценки
fSredn_Ocenka= 0; // Очистка ячейки
for (int ij= 0; ij< m; ++ij)
    fSredn_Ocenka += iOcenka[ij];

//Определяем среднюю оценку
fSredn_Ocenka= fSredn_Ocenka/m;

//Определяем минимальную оценку
iMin_Ocenka= iOcenka[0];
for (int ij= 1; ij< m; ++ij)
    if (iOcenka[ij]< iMin_Ocenka)
        iMin_Ocenka= iOcenka[ij];

//Определяем число учащихся, у которых оценка больше 7
iBolshe_7= 0; // Очистка ячейки
for (int ij= 0; ij< m; ++ij)
    if (iOcenka[ij]> 7)
        iBolshe_7++;

//Вывод результатов
Sredn_Ocenka->Caption= "Средняя оценка =" + FloatToStrF(fSredn_Ocenka,
ffFixed, 10, 2);
Min_Ocenka->Caption= "Минимальная оценка =" + IntToStr(iMin_Ocenka);
Bolshe_7->Caption= "Число оценок превышающих 7 равно " +
IntToStr(iBolshe_7);
```

Показ итоговых результатов поясним на примере вывода значения *fSredn_Ocenka*.

```
Sredn_Ocenka->Caption= "Средняя оценка = " +  
    FloatToStrF(fSredn_Ocenka, ffFixed, 10, 2);
```

Здесь к текстовой константе «*Средняя оценка =*» при помощи знака (+) конкатенации присоединяется преобразованное к текстовому виду значение величины *fSredn_Ocenka*. Текстовым переменным и функциям для их обработки посвящён двенадцатый урок. Здесь же лишь отметим, что текстовые переменные и константы можно сцеплять между собой (присоединять друг к другу) при помощи знака «+».

6.1.7. Компактный вариант функции *PyskClick*

Выше для ясности изложения каждый из алгоритмов обработки массивов был рассмотрен по отдельности, поэтому цикл *for* в функции *PyskClick* применялся 4 раза. Теперь приведём вариант тела этой же функции с использованием только одного цикла.

```
const int m= StrToInt(Ch_Ych->Text); //Текущее число учащихся  
float fSredn_Ocenka= 0; //Средняя оценка  
int iMin_Ocenka, //Минимальная оценка  
iBolshe_7= 0; //Число учащихся, у которых оценка больше 7  
int iOcenka[n]; //Массив оценок  
for (int ij= 0; ij< m; ++ij)  
{  
    iOcenka[ij]= StrToInt(Tab->Cells[1][ij + 1]); //Ввод данных  
    if (ij== 0)  
        iMin_Ocenka= iOcenka[0];  
    fSredn_Ocenka += iOcenka[ij]; //Суммируем оценки  
    if (iOcenka[ij]< iMin_Ocenka) //Определяем минимальную //оценку  
        iMin_Ocenka= iOcenka[ij];  
    if (iOcenka[ij]> 7) //Определяем число оценок больше 7  
        iBolshe_7++;  
} //for_ij  
//Определяем среднюю оценку  
fSredn_Ocenka= fSredn_Ocenka/m;  
//Вывод результатов  
Sredn_Ocenka->Caption= "Средняя оценка = " + FloatToStrF(fSredn_Ocenka,  
ffFixed, 10, 2);  
Min_Ocenka->Caption= "Минимальная оценка = " + IntToStr(iMin_Ocenka);  
Bolshe_7->Caption= "Число учащихся с оценкой выше 7 равно " +  
IntToStr(iBolshe_7++);
```

Приведенный вариант тела функции не только более компактен, но и выполняется за значительно меньшее время.

Вопрос для текущего самоконтроля

- ☐ Оцените, во сколько раз последний вариант тела функции выполняется быстрее, чем предшествующий вариант?

6.1.8. Общие определения

Массив – это тип данных, состоящий из *фиксированного* числа *однотипных* элементов. Число элементов и их тип фиксируются при описании статического массива и в процессе выполнения программы не изменяются (это правило не относится к динамическим массивам, о них будет рассказано на десятом уроке). Массивы предназначены для хранения *целого набора* данных *одинакового* типа и для *удобного доступа* к содержимому каждого элемента этого набора.

Для описания массива используется его имя – идентификатор, впереди которого приводится тип элементов массива, а позади него в квадратных скобках указывается число элементов массива. Начальный индекс массива является нулём.

При разработке программ иногда возникает необходимость объявленные массивы сразу же инициализировать конкретными значениями. Так, например, будет выглядеть фрагмент функции *PyskClick*, в которой массив *iOcenka[n]* после объявления сразу же инициализируется оценками ведомости.

```
int iOcenka[n]= {5, 3, 12, 9, 8, 6, 10, 11, 7, 8};
```

Из примера видно, что инициализация осуществляется при помощи знака присваивания и фигурных скобок, внутри которых через запятую перечисляются инициализирующие значения (в нашем случае это оценки ведомости успеваемости группы учащихся). Первое значение (5) записывается в нулевую ячейку, второе (3) – в первую и так далее. Число инициализирующих значений *может быть меньше* размера массива. В этом случае в оставшихся ячейках локального массива окажутся неопределённые значения, а в незанятых ячейках глобального массива – нулевые значения. Если же вы попытаетесь в фигурных скобках перечислить значения, число которых *превышает* размер массива, то *Builder* на этапе компиляции программы выведет сообщение о допущенной ошибке: «E2225 Too many initializers» (слишком много инициализирующих значений).

Можно задать и инициализировать массив ещё и таким способом:

```
int iOcenka[] = {5, 3, 12, 9, 8, 6, 10, 11, 7, 8};
```

Если в квадратных скобках не указывать размер массива, то число его элементов задаётся числом инициализирующих значений. Такую возможность определения размерности массива не рекомендуем использовать, поскольку она не способствует беглому чтению текста программы.

Применение статических массивов связано со следующими ограничениями:

- ❑ Число элементов массива (или размерность массива) задаётся *целой константой* или константным выражением, результатом которого является *целое* значение. Число элементов массива должно быть больше нуля.
- ❑ Максимально возможное число *n_Lokaln* элементов *локального* массива определяется по формуле

$$n_Lokaln = Razmer_Steka/p;$$

- ❑ Максимально возможное число *n_Globaln* элементов *глобального* массива определяется выражением

$$n_Globaln = Razmer_Svobodn_Operat_Pam/p;$$

здесь p – размер элемента массива в байтах, $Razmer_Steka$ – размер стека в байтах (оперативная память, выделяемая на время работы функции для размещения всех её переменных и кода), $Razmer_Svobodn_Operat_Pam$ – размер свободной оперативной памяти компьютера в байтах.

Размер стека по умолчанию составляет 1 МБ, поэтому максимальное число элементов локального массива типа *int* и *float* составляет 262 144 (1 МБ = 1 024 * 1 024 байт = 1 048 576 байт, 1 048 576 байт / 4 = 262 144), а типа *doubl* – 131 072 (1 048 576 байт / 8 = 131 072).

Если же массив объявлен глобальным (способы объявления см. в следующем пункте), то его максимальный размер для оперативной памяти, например, в 512 МБ, будет в 512 раз больше. Программисты, использующие компиляторы под управлением операционной системы *MS-DOS*, не могут даже мечтать о таких больших массивах: максимальный объём глобального статического массива, например, в *TurboPascal* не может превышать объём сегмента данных в 64 кБ (при использовании динамической памяти – до ~400 кБ).

6.1.9. Глобальные переменные и константы

Объявить переменные (в том числе и массивы) глобальными можно тремя способами.

1. Открыть файл, содержащий код описания класса основной формы проекта. Для этого следует перейти в *Редактор Кода*, ввести команду *Ctrl+F6* (в *Builder 6* можно также воспользоваться ярлычком этого файла с расширением *h*). Перенестись в этот файл можно и с использованием контекстного меню: щёлкните *правой* кнопкой мыши по свободной части окна *Редактора Кода* и активизируйте пункт появившегося меню *Open Source/Header File* (Открытие источника/Заголовочный файл). Затем, в разделе класса формы *public* (общий или открытый) с комментарием *User declarations* (объявления пользователя), сделайте объявление глобального массива, например: «*int iOcenka[100];*». Инициализировать объявляемые переменные в этом месте файла *нельзя*. *Запрещается* здесь также объявлять *константы*. Однако вне описания класса (за строкой *#endif* или непосредственно перед ней) разрешается инициализировать глобальные переменные и вводить глобальные константы. Настоятельно *не рекомендуем* на первых шагах обучения программированию что-либо изменять в этом файле, он генерируется автоматически, подробно его устройство рассматривается на четырнадцатом уроке.
2. Открыть головной файл проекта (в нём содержится основная функция проекта *WinMain*): ввести команду: *Project/View Source*. Глобальные константы и переменные объявляются в конце этого файла, объявленные переменные разрешается инициализировать. Объявления констант и переменных необходимо повторить и в файле реализации, где расположена

функция, в которой эти константы и переменные используются. Начинаться объявления должны с ключевого слова *extern* (внешний). Описание в головном файле проекта может выглядеть, например, так:

```
const int n= 100; //Максимальное число учащихся
int iOcenka[n]= {5, 3, 12, 9, 8}; //Массив оценок
```

В файле реализации (в этом файле (с расширением *«.cpp»*) находятся функции, например, *PyskClick* и *FormCreate*, которые применялись ранее), где эти константа и массив используются, следует сделать такое объявление:

```
extern const int n= 100; //Максимальное число учащихся
extern int iOcenka[n]; //Массив оценок
```

Эти объявления означают, что указанные константа и массив введены *вне* настоящего файла. При этом переменным и константам, объявленным и инициализированным в головном файле, в файле реализации *могут* быть назначены и *новые* значения.

3. Глобальные переменные и константы объявить в файле реализации.

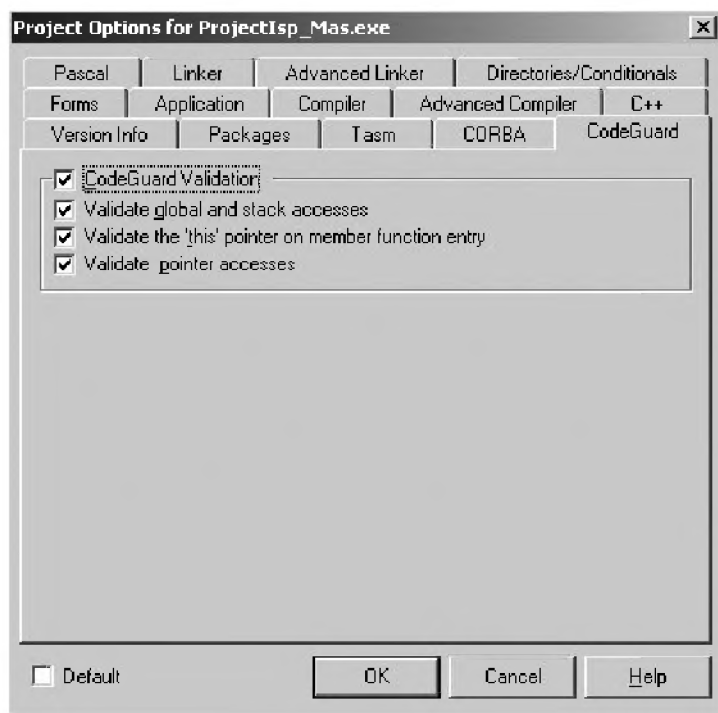
Настоятельно рекомендуем до четырнадцатого урока глобальные переменные и константы объявлять только третьим способом.

6.1.10. Страж кода

С целью повышения быстродействия программ *Builder* по умолчанию не контролирует выход номера элемента массива за его границы. Поэтому если в ходе работы программы попытаться обратиться к элементу массива, индекс которого находится *за границами*, указанными при определении массива (меньше нуля или больше $(n - 1)$), то *Builder* не сообщит о допущенной ошибке: вернёт какие-то значения, хранящиеся на этот момент в соответствующих ячейках памяти. Интерпретация двоичных значений, размещённых в этих ячейках, будет зависеть от типа (данных) элементов упомянутого массива.

Из запредельных ячеек массива можно не только прочесть значения, но и записать в них какие-то данные. Вариант с записью наиболее опасен. Ведь эти данные могут располагаться (полностью либо частично) в ячейках, занятых *другими* переменными вашей программы, иных работающих приложений, самого *Builder* или даже операционной системы. В последнем случае компьютер может зависнуть.

Вся ответственность за появление указанных ошибок полностью *лежит на разработчике* программ. Те программисты, которые тяготеют такой ответственностью (на этапе разработки проектов) могут воспользоваться услугами *Стража Кода* (*CodeGuard*). *Страж Кода* контролирует выход за выделенные при объявлении границы массива. Его подключение осуществляется командой: *Project/Option* (*Ctrl+Shift+F11*), вкладка *CodeGuard*. Далее на выведенной вкладке (см. рис. 6.2) включите *все* флажки (они определяют подключение *CodeGuard*), а затем перекомпилировать проект командой *Project/Build*.

Рис. 6.2. Вкладка параметров *CodeGuard*

Дополнительная настройка параметров *CodeGuard* (его загрузка в разрабатываемый проект) осуществляется командой *Tools/CodeGuard Configuration* (Сервис/Конфигурация *CodeGuard*). В течение секунды или более (это время зависит от тактовой частоты процессора вашего компьютера) *Страж* загрузится в разрабатываемый проект, и в диалоговом окне (см. рис. 6.3) появится возможность изменить его параметры, в разделе *CodeGuard is Enabled* флажок *Enable* (включено) является активизированным. При первом знакомстве настоятельно рекомендуем конфигурацию по умолчанию *не изменять*: щёлкните по кнопке *OK*. Подключить *CodeGuard* можно *только* к *отдельному* проекту, при разработке нового приложения *Страж* вновь по умолчанию будет отключён.

К концу настоящего раздела читатели самостоятельно могут дать правильный ответ на вопрос, сформулированный в его названии: «Зачем нужны массивы?». Да, массивы предназначены для хранения и удобной обработки табличных данных.

На этом уроке рассматриваются линейные или одномерные массивы – они выглядят в виде одной строки или столбца. На следующем уроке познакомимся с массивами больших размерностей, в частности двумерными и трёхмерными. Они содержат наборы строк и столбцов. Массивы, состоящие из совокупности элементов, являются примером переменных *структурированного* типа. К таким переменным относятся также *структуры* (*struct*), *перечисления* (*enum*) и *объединения* (*union*). О них рассказывается на тринадцатом уроке.

Ниже продолжим накапливать приёмы обработки *одномерных* массивов. Помимо хранения, ввода и вывода на экран, суммирования их элементов, подсчёта количества компонент, значения которых попадают в заданный интервал, важным пунктом обработки массивов являются различные методы упорядочивания их компонент. Одному из так методов посвящён наш следующий раздел.

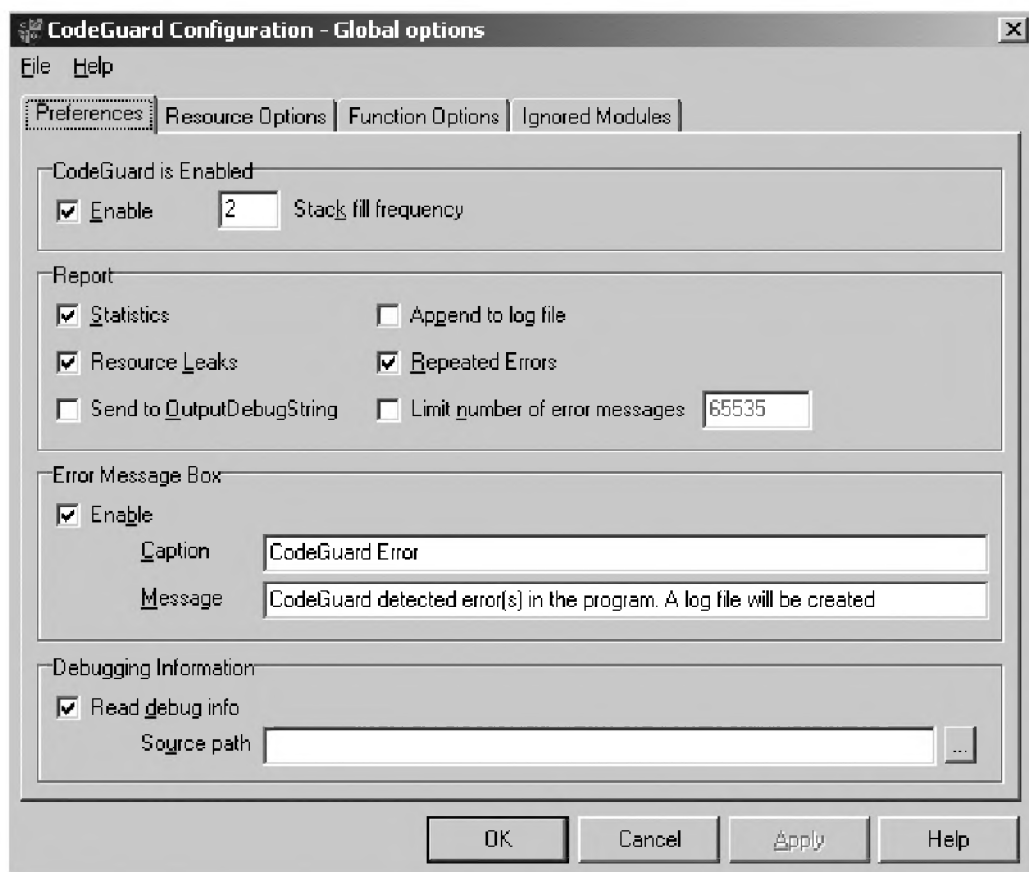


Рис. 6.3. Настройка параметров CodeGuard

6.2. Метод линейной сортировки

6.2.1. Сущность метода

Учёт и контроль правят бал практически во всех областях деятельности человека. Однако эффективно исполнять эти очень важные действия возможно лишь после процесса упорядочивания. Только после приведения в порядок, например, книг личной библиотеки вы с удивлением можете обнаружить, что целый ряд дорогих вашему сердцу изданий отсутствуют на полке и вместе с тем чужие книги, своевременно не возвращённые их хозяевам, сиротливо собирают квартирную пыль.

Существуют различные методы упорядочивания, каждый из них основан на каком-то правиле. В библиотеках имеются алфавитные и систематические каталоги, фамилии в списках располагают также по алфавиту, а вот на уроках физкультуры школьники выстраиваются по росту. Такие и подобные им пути борьбы с беспорядком относятся к методам упорядочивания.

Борьбу за всеобщий порядок начнём с изучения метода упорядочивания или сортировки элементов массива $R[n]$ по возрастанию значений. Пусть этот массив будет небольшим, состоять всего лишь из пяти элементов:

15 106 12 5 22.

Справиться с ручной сортировкой его элементов очень легко, в результате

получится возрастающий ряд чисел:

5 12 15 22 106.

Имеется ряд методов выполнения такого упорядочивания. Думаем, читатель интуитивно выберет метод, наиболее простой для *ручной* сортировки. Его сущность заключается в следующих действиях. В исходном массиве перечеркнуть наименьший элемент (5) и записать его значение на первой позиции новой строки, расположенной под первоначальным набором чисел. Далее, из оставшихся членов сортируемого ряда вновь выбрать и зачеркнуть наименьшее число (12), записав его во вторую позицию новой строки. Продолжая этот процесс дальше, вы обнаружите, что последнее, ещё не перечёркнутое число (106) в исходном массиве, является максимальным и его, безусловно, следует поставить последним в нижней строке. Этим действием успешно завершается процесс ручной сортировки.

При ручной сортировке это, пожалуй, неплохой метод, который позволяет без ошибок выполнить требуемое условием упорядочивание. Однако массивы большой длины не рекомендуем подвергать *ручной* сортировке. Такая скучная неинтересная утомительная работа – удел компьютерных программ, они с большим удовольствием правильно и быстро с нею справляются. И такого рода программ существует превеликое множество. Даже *текстовый* редактор *Word*, в котором набирались эти строки, умеет её выполнять. Что же касается иных офисных программ *Excel* и *Access*, то упомянутые услуги входят в круг их *непосредственных* обязанностей.

Но вы, дорогой читатель, – не простой пользователь *уже готовых* программ, вы приобщились к славной когорте разработчиков подобных продуктов, поэтому должны научиться программировать операции сортировки *самостоятельно*. Только для упомянутых целей не используйте описанный алгоритм ручной сортировки, он для их реализации не годится. И не потому, что он неверен, нет, вы убедились в том, что он приводит к правильному результату. Дело в том, что его реализация потребовала применения *двух* строк для записи (хранения) массивов одинаковой длины (исходного и отсортированного). Следовательно, в программе согласно этому алгоритму для упорядочивания *одного* массива необходимо выделить *две* одинаковые переменные-массивы. Однако в программировании такое расточительство недопустимо. Ведь длина массива может быть очень большой и вследствие этого такая программа окажется не всегда эффективной. Поэтому ниже познакомимся с методом сортировки, не использующим дополнительный второй массив.

6.2.2. Алгоритм метода

Шаги этого алгоритма можно сформулировать в следующей редакции.

1. Найти минимальный элемент массива и поместить его в $iR[0]$. Это достигается путём сравнения $iR[0]$ со всеми последующими элементами массива (то есть с $iR[1]$, $iR[2]$, $iR[3]$, $iR[4]$). В случае, когда для сравниваемых пар элементов массива неравенство $iR[0] < iR[ij]$ нарушается, значения

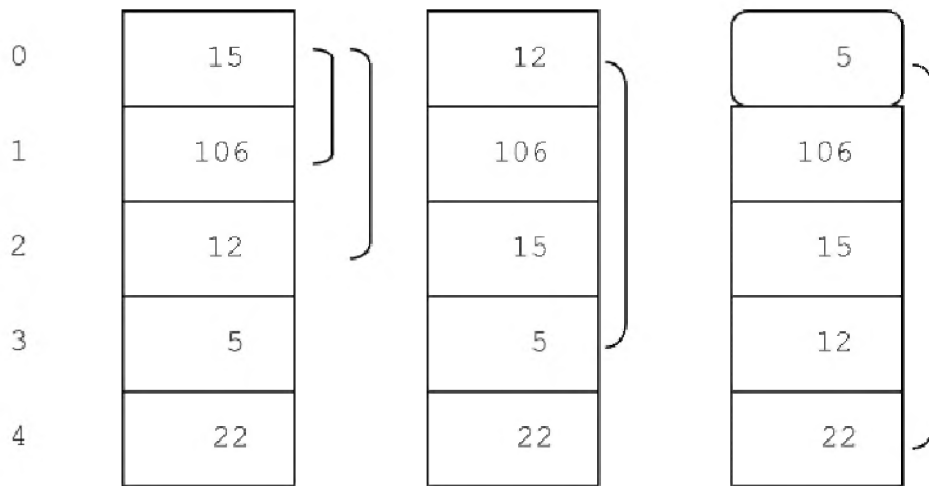


Рис. 6.4. Сравнение $iR[0]$ с другими элементами массива

этих ячеек меняются местами и процесс сравнения выполняется дальше вплоть до последней ячейки.

- Шаг 1 повторить для элементов массива, начиная с первого: среди оставшихся элементов найти наименьший и поместить его в $iR[1]$.
- Повторить шаг 1, начиная со второго, затем третьего, кроме четвертого – последнего элемента массива.

6.2.3. Ручная трассировка алгоритма

Изложенный алгоритм подвергнем ручной пошаговой проверке или трассировке. Такую трассировку произведём над прежним массивом $iR[n]$. Для удобства анализа его элементы пронумеруем и их значения разместим в виде столбца прямоугольных ячеек. Сравнимые элементы будем указывать концами скобок. После каждой перестановки новое расположение элементов массива приводится в соседнем столбце. Отобранные элементы массива отмечены скруглёнными прямоугольниками. Они уже заняли своё место в отсортированной части массива, и поэтому исключаются из дальнейшего рассмотрения.

Все этапы выполнения первого шага алгоритма иллюстрируются протоколом перестановок, показанным на рис. 6.4.

В ходе поиска минимального элемента здесь осуществляется цикл сравнения $R[0]$ со всеми последующими элементами массива. В результате двух перестановок минимальный элемент (5) занял свою искомую нулевую позицию, однако все остальные элементы остались неупорядоченными. Необходимо поэтому перейти к выполнению второго шага алгоритма – второго цикла сравнения (см. рис. 6.5).

Отметим, что ко второму и последующим шагам алгоритма требуется переходить *всегда*, невзирая на то, упорядочены или не упорядочены оставшиеся элементы массива. Дело в том, что этот алгоритм предназначен для *компьютера*, который может правильно решить поставленную задачу, лишь *последовательно* выполняя *все* свои шаги.

Для упорядочивания второго элемента массива также потребовалось две пе-

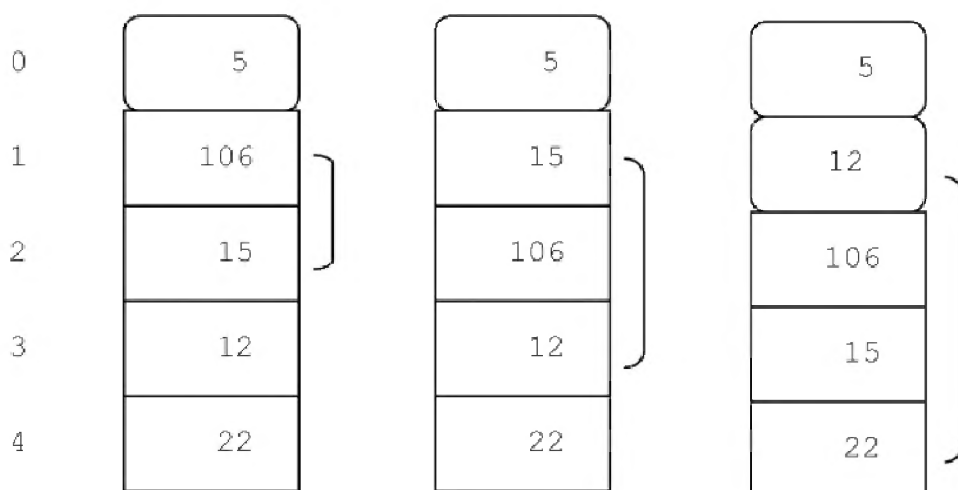


Рис. 6.5. Второй цикл сравнения

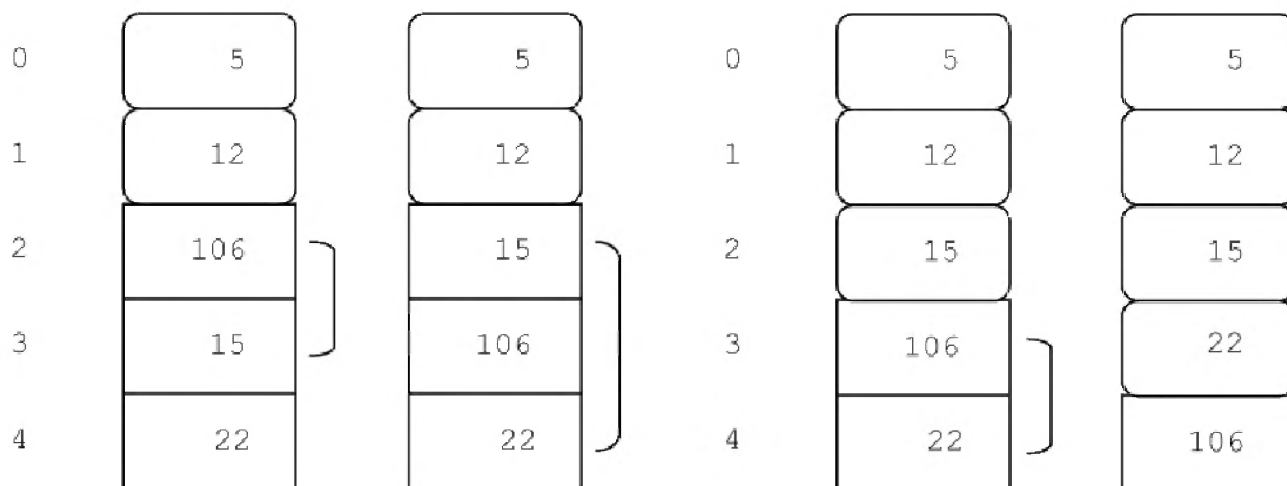


Рис. 6.6. Третий цикл сравнения

Рис. 6.7. Четвертый цикл сравнения

рестановки элементов, а вот третий цикл сравнения справился со своей работой, в результате лишь одной перестановки (рис. 6.6).

Протокол сравнений и перестановок последнего, четвёртого, цикла сравнения показан на рис. 6.7. Как видно, в последнем цикле может быть выполнено *только одно* сравнение, для использованных данных понадобилась также и перестановка сравниваемых элементов. Но в других реализациях массива она может оказаться и ненужной.

Рассмотренный массив имеет 5 элементов, для его *полной* сортировки потребовалось 4 цикла сравнения. Сколько же циклов понадобится для упорядочивания массива из 6 элементов? Согласитесь, что в этом случае следует выполнить уже 5 циклов. Линейная сортировка массива, состоящего из n элементов, требует выполнения $(n - 1)$ циклов сравнения.

В описанном алгоритме на каждом шаге производится отбор одного единственного элемента, и поэтому этот алгоритм часто называют сортировкой методом отбора. Название же метода *линейная сортировка* происходит от последовательного просмотра всех элементов массива, находящихся в их исходном *линейном* порядке.

6.2.4. Иллюстрация метода

Разработаем приложение, в котором при помощи метода линейной сортировки упорядочиваются по возрастанию значений элементы массива. На рис. 6.8 показан интерфейс программы с исходными и итоговыми значениями массива. Здесь таблица строк названа *Tab*, поле ввода – *Vvod_Dl_Mas*, а кнопка – *Pysk*.

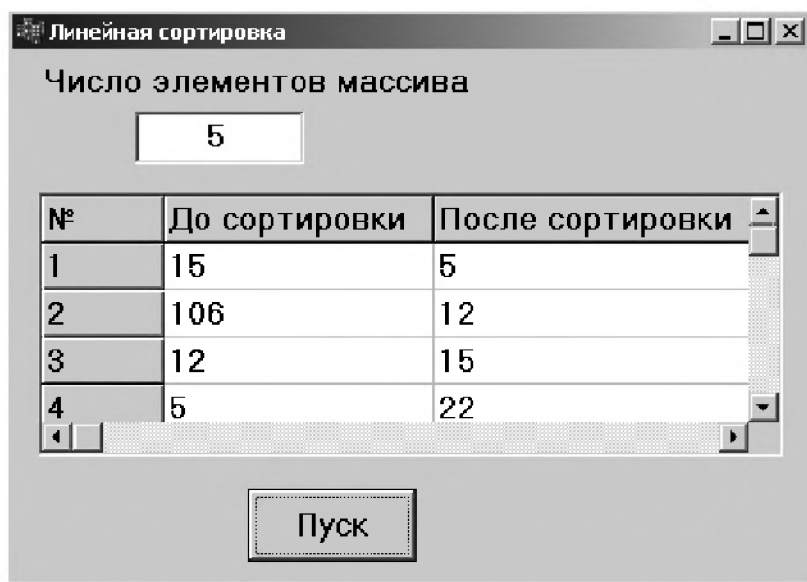


Рис. 6.8. Интерфейс приложения *Линейная сортировка*

В файле реализации введены глобальная константа и массив:

```
const int n= 100; //Максимальная длина массива
int iR[n]; //Массив
```

Код функции *FormCreate* имеет вид:

```
//Текущая длина массива
const int m= StrToInt(Vvod_Dl_Mas->Text);
//Текущее число строк таблицы
Tab->RowCount= m + 1;
//Заголовки столбцов таблицы
Tab->Cells[0][0]= "№";
Tab->Cells[1][0]= "До сортировки";
Tab->Cells[2][0]= "После сортировки";
//Нумерация всех возможных строк таблицы
for (int ij= 1; ij<= n; ++ij)
    Tab->Cells[0][ij]= ij;
```

Тело функции *Vvod_Dl_MasChange* обеспечивает порождение указанного числа строк таблицы, вертикальный размер окна вывода таблицы (свойство *Height*) согласует с числом строк, если их не больше 6 и устанавливает равным 6 в противном случае (при этом появляется вертикальная полоса прокрутки).

```
//Текущая длина массива
const int m= StrToInt(Vvod_Dl_Mas->Text);
Tab->RowCount= m + 1;
if (m>= 6)
```



```
Tab->Height= 26*6;
else
    Tab->Height= 26*(m + 1);
```

Ввод и вывод данных в таблицу строк освоен в предшествующих приложениях. Поэтому далее сосредоточим внимание только на программной реализации метода линейной сортировки.

В этом методе при отборе каждого ij -го элемента массива требуется выполнять его сравнение с последующими элементами, имеющие номера: $ij + 1, ij + 2, ij + 3, \dots, ij + n - 1$. И каждый из них может оказаться текущим кандидатом искомого элемента упорядочиваемого ряда. Поэтому переменную, обозначающую номер такого дежурного кандидата, назовём $iCan$ (от *candidat*).

Поскольку число сопоставлений в каждом цикле отбора минимального значения всегда известно, то удобнее всего применить цикл *for*, переменная $iCan$ будет играть в нём роль параметра цикла.

```
for (int iCan= ij + 1, iByf; iCan< m; ++iCan)
    if (iR[ij]> iR[iCan])
    {
        // Меняем местами содержимое ячеек
    }//if
```

Здесь логический блок оператора *if* осуществляет перестановку элементов $iR[ij]$ и $iR[iCan]$. Если на место элемента $iR[ij]$ поместить элемент $iR[iCan]$, то прежнее значение в ячейке $iR[ij]$ исчезнет навсегда; если же, наоборот, ячейке $iR[iCan]$ присвоить величину из $iR[ij]$, то окажется потерянным значение в $iR[iCan]$. Однако такие потери недопустимы, поскольку и $iR[ij]$, и $iR[iCan]$ должны быть соответствующими элементами отсортированного массива. Поэтому для временного хранения $iR[ij]$ (или $iR[iCan]$) в разделе инициализации цикла *for* введена переменная $iByf$ (от *Byfer*). В этом случае перестановку элементов можно организовать следующей последовательностью операторов:

```
iByf= iR[ij];
iR[ij]= iR[iCan];
iR[iCan]= iByf;
```

Число циклов сравнения также полностью определяется количеством элементов массива, поэтому было бы менее разумным для изменения номера отбираемого элемента применять вместо цикла *for* иной цикл. После упомянутых предварительных разъяснений покажем в законченном виде, как программным путём осуществляется линейная сортировка массива iR .

```
for (int ij= 0; ij< m - 1; ++ij)
    for (int iCan= ij+1, iByf; iCan< m; ++iCan)
        if (iR[ij]> iR[iCan])
        {
            //Меняем местами содержимое ячеек
            iByf= iR[ij];
            iR[ij]= iR[iCan];
            iR[iCan]= iByf;
        }//if
```

Внешний цикл *for_ij* в этом фрагменте $(m - 1)$ раз запускает на выполнение внутренний цикл *for_iCan*. При этом осуществляется $(m - 1)$ циклов сравнения, о которых говорилось выше. Внутренний же цикл *for_iCan*, как уже отмечалось, сравнивает элемент *iR[ij]* со всеми оставшимися элементами до конца массива (до элемента с номером $(m - 1)$).

Вопрос для текущего самоконтроля

- ❑ Почему во внешнем цикле сравнение выполняется до элемента с номером $(m - 2)$?

Осуществим *пошаговый* анализ работы этого фрагмента. При *ij* = 0 внутренний цикл *for_iCan* сравнивает *iR[0]* со всеми другими компонентами массива от *iR[1]* до *iR[m - 1]* (см. первый шаг алгоритма). В результате минимальный элемент массива оказывается в нулевой ячейке *iR[0]*.

При втором обороте внешнего цикла, когда *ij* = 1, сравниваются содержимое ячейки *iR[1]* с обитателями всех последующих ячеек от *iR[2]* до *iR[m - 1]* (см. второй шаг алгоритма). В этом случае уже законный претендент на второе место в массиве *iR* радостно разместит свои пожитки во второй его камере с номером *iR[1]*.

Последний шаг внешнего цикла *for_ij* выполняется при *ij* = $m - 2$, при этом в его внутреннем цикле сравнивается лишь одна пара значений: *iR[m - 2]* и *iR[m - 1]* (см. третий шаг алгоритма). Однако это обстоятельство никак не препятствует самому весомому кандидату оказаться в последней ячейке массива и завершить таким образом процесс упорядочивания элементов массива *iR* в порядке возрастания их значений.

Сравнение результатов пошагового анализа приведенного фрагмента и результатов ручной трассировки алгоритма (см. п. 6.2.3) указывает на их полное совпадение. Это доказывает правильность работы программной реализации алгоритма линейной сортировки. Именно к такому, пошаговому ручному, разбору программных кодов следует всегда прибегать для проверки правильности их функционирования. Не спешите запускать их на выполнение, и ваша выдержка с торией окупится, уверяем вас!

Обсуждаемый алгоритм, конечно, можно реализовать и с использованием циклов *while* или *do_while*. Однако такие хитрости лишь затуманят прозрачность кода, поэтому настоятельно рекомендуем их избегать: «*Выражайтесь проще, Шура, и к вам потянутся люди*».

Ниже приводится полный текст функции *PyskClick*, которая после нажатия кнопки *Пуск* обеспечивает ввод, сортировку и вывод массива.

```
//Текущая длина массива
const int m= StrToInt(Vvod_Dl_Mas->Text);
//Заносим исходные данные в массив iR[n]
for (int ij= 1; ij<= m; ++ij)
    iR[ij - 1]= StrToInt(Tab->Cells[1][ij]);
//Линейная сортировка
for (int ij= 0; ij< m - 1; ++ij)
    for (int iCan= ij + 1, iByf; iCan< m; ++iCan)
```

```
if (iR[ij] > iR[iCan])
{
    //Меняем местами содержимое ячеек
    iByf = iR[ij];
    iR[ij] = iR[iCan];
    iR[iCan] = iByf;
} //if
//Вывод массива
for (int ij = 0; ij < m; ++ij)
    Tab->Cells[2][ij + 1] = iR[ij];
```

Вопрос для текущего самоконтроля

- Что необходимо изменить в функции *PyskClick* для того, чтобы онаставляла элементы массива по убыванию их значений?

6.3. Сортировка методом пузырька

Как уже отмечалось, существует целый ряд методов, предназначенных для распределения элементов по возрастанию или убыванию их значений. Эти методы отличаются своим быстрым действием. Для первого знакомства выбраны наиболее простые из них (для изложения и понимания), большие массивы данных они упорядочивают слишком медленно. Но скорость сортировки небольших массивов у простых и сложных методов приблизительно одинакова, в некоторых случаях простой метод работает даже быстрее, чем его замысловатый брат. Поэтому простые методы разумно применять не только в учебных целях.

К группе простых методов относится также пузырьковая сортировка. Для рассмотрения алгоритма метода пузырька проанализируем вначале структуру полностью *отсортированного* массива. Пусть это будет прежний массив *iR*: 5; 12; 15; 22; 106. Как устроен отсортированный массив? В *каждой паре* смежных элементов такого массива последующий элемент больше предыдущего: $iR[0] < iR[1]$, $iR[1] < iR[2]$, $iR[2] < iR[3]$, ..., $iR[n - 2] < iR[n - 1]$. Поэтому можно сделать достаточно очевидный вывод: если каждый элемент массива меньше непосредственно следующего за ним элемента, то этот массив полностью отсортирован.

Указанный индикатор упорядоченности позволяет организовать сортировку массива следующим способом: последовательно рассматривать каждую пару его элементов и если последующий элемент окажется больше предыдущего, то перейти к анализу следующей пары, если же неравенство нарушается, то сравниваемые элементы необходимо поменять местами и продолжать такие попарные сравнения вплоть до последней пары.

Выполнив сравнение и возможную перестановку элементов последней пары массива, нельзя быть уверенным в том, что массив оказался полностью отсортированным. Дело в том, что если была произведена перестановка значений, например, $iR[2]$ и $iR[3]$, то может случиться так, что новое значение $iR[2]$ (полученное из $iR[3]$) будет меньше, чем $iR[1]$. Поэтому требуется повторять попарную проверку массива до тех пор, пока не заработает индикатор упорядоченности: при *очередном* цикле сравнения не произойдет *ни одной* перестановки смежных пар.

После такого сигнала сортировка завершается.

Таким образом, новый метод сортировки состоит из циклов попарного упорядочивания соседних элементов, которые производятся до тех пор, пока не будет обнаружен цикл, в котором не было произведено *ни одной* перестановки смежных элементов.

Для программной реализации этого метода несколько формализуем его алгоритм:

1. Для элементов массива проверить выполнение последовательности неравенств: $iR[0] < iR[1]$, $iR[1] < iR[2]$, $iR[2] < iR[3]$, ..., $iR[n - 2] < iR[n - 1]$. В случае нарушения неравенства, сравниваемые элементы поменять местами.
2. Повторять выполнение шага 1, пока не будет обнаружен цикл, в ходе которого не будет зарегистрирован ни один случай нарушения неравенств.

На рис. 6.9 и 6.10 приведены протоколы ручной трассировки алгоритма пузырькового метода – иллюстрации сравнений и перестановок элементов сортируемого массива. Для такого пошагового анализа использован прежний числовой массив и обозначения: сравниваемые элементы массива отмечаются скобкой, в новом столбце записываются новые положения элементов массива после их перестановки.

Метод получил название *метод пузырька*, поскольку в нём минимальное число массива (5) перемещается в начало массива, аналогично пузырьку воздуха, всплывающему на поверхность воды.

Если исходный массив полностью отсортирован, то метод пузырька выяснит это в результате первого же цикла сравнения. Линейному методу для сортировки такого массива потребуется $(n - 1)$ цикл сравнения. В методе линейной сортировки число шагов внешнего цикла определяется *только* длиной исходного массива и *не зависит* от степени у его упорядоченности. Вместе с тем, чем более упорядочен исходный массив, тем за меньшее время пузырьковый метод справится с его окончательной сортировкой. В этом методе число циклов сравнения может изменяться от 1, для полностью упорядоченных массивов, до n , когда все элементы массива стоят в обратном порядке. В общем случае, когда массивы лишь немного упорядочены, эффективность обоих методов примерно одинакова, для частично отсортированных массивов эффективнее применять метод пузырька.

Ниже показан фрагмент функции *PyskClick*, реализующий сортировку массива методом пузырька (другие части функции не изменилась). После кода этого фрагмента приводится подробное рассмотрение особенностей его работы. Интерфейс программы отличается от интерфейса приложения *Линейная сортировка*, только его названием *Сортировка методом пузырька*. Функции *FormCreate*, *Vvod_Dl_MasChange* остались прежними.

```
//Осуществляем сортировку методом пузырька
bool Sort;
do
{
    Sort= false;
    for (int ij= 0, iByf; ij< m - 1; ++ij)
```

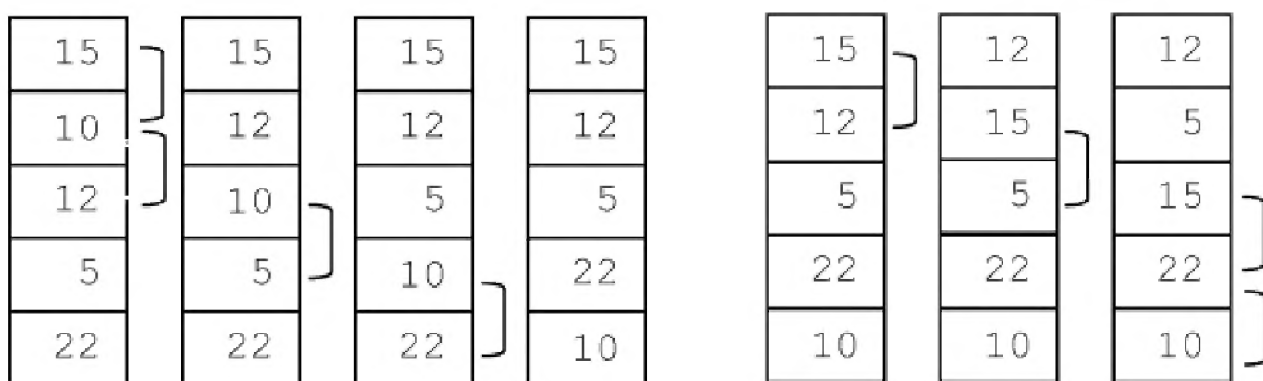


Рис. 6.9. Первый и второй циклы

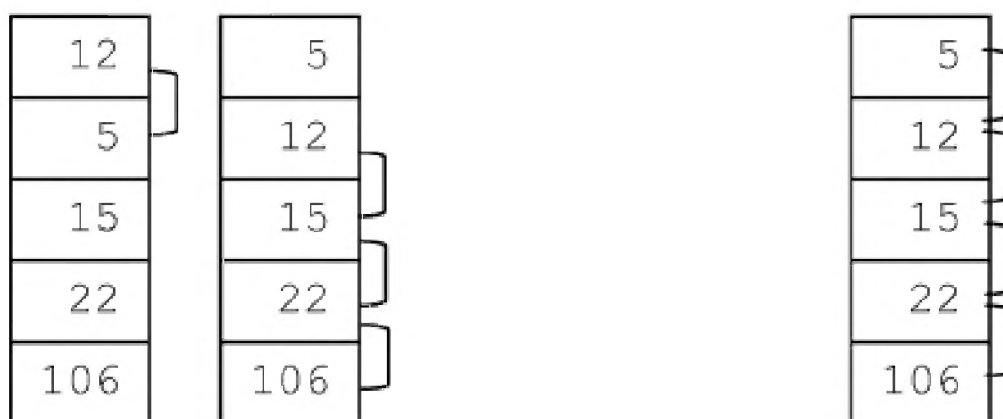


Рис. 6.10. Третий и четвертый циклы

```

if (iR[ij]> iR[ij + 1])
{
    // Меняем местами содержимое ячеек
    iByf= iR[ij];
    iR[ij]= iR[ij+1];
    iR[ij + 1]= iByf;
    Sort= true; //Индикатор перестановки
} //if
} // do
while(Sort);

```

while. Если такое условие не выполняется (*Sort == true*), то его внутренний цикл запускается на выполнение ещё раз, в противном случае управление программой передаётся оператору, стоящему за строкой «*while(Sort);*».

Таким образом, возобновление работы внутреннего цикла по сортировке элементов массива определяется событием: была перестановка сравниваемых пар компонент. Контроль за событием: *осуществилась перестановка* выполняет переменная *Sort* логического типа *bool*. Перед каждым циклом попарного сравнения элементов массива (перед циклом *for_ij*) этой переменной присваивается значение *false*. Если в цикле неравенство ($R[ij] > R[ij + 1]$) выполнится хотя бы один раз, то в конце этого цикла в переменной *Sort* окажется значение *true*. Такое значение этой переменной даёт указание внешнему циклу *do_while* запустить свой внутренний цикл сравнения *for_ij* ещё раз. Если же такой сигнал-указание не последует, то цикл *do_while* завершает свою работу, массив является отсортированным.

Внешним циклом может быть также цикл *for* в содружестве с оператором досрочного выхода из цикла *break* (см. раздел 4.2). Однако более изящным решением является всё же применение цикла, специально предназначенного для работы в таких условиях: цикла *do_while*.

Можно применить вариант, в котором перед циклом *for_ij* переменной *Sort* присваивается значение *true* (а не *false*), тогда в теле оператора *if* в эту переменную записывается *false*. В этом варианте постусловие должно быть таким: *while(Sort == false);* или же таким: *while(!Sort);*. Последний вариант реализации цикла нам представляется менее наглядным.

6.4. Другие методы упорядочивания элементов массива

Преобразуем расположение элементов целочисленного массива так, чтобы в начальные его ячейки были перемещены все нечетные элементы, а все четные компоненты располагались бы после них, в оставшихся ячейках. Существует целый ряд вариантов решения этой задачи.

6.4.1. Применение трёх массивов

В первом из них для этой цели используем три переменных-массива с заданной длиной *n* и типом *int* их компонент. В первую такую переменную поместим исходный массив, во вторую и третью скопируем соответственно нечётные и чётные элементы исходного массива. Затем объединим нечётный и чётный массивы в один массив. Интерфейс этого приложения такой же, как в рассмотренных программах сортировки (за исключением заголовка). Функции *FormCreate* и *Vvod_Dl_MasChange* также не претерпели изменений.

В функции *PyskClick* из прежнего приложения фрагменты, выполняющие сортировку и вывод результатов преобразования исходного массива, замените кодом, который приведен ниже.

```

int in_Ch_El= 0, // Количество чётных элементов
in_NCh_El= 0, //Количество нечётных элементов
iCh[n], // Массив для хранения чётных чисел
iNCh[n]; // Массив для хранения нечётных чисел
//Осуществляем сортировку
for (int ij= 0; ij< m; ++ij)
{
    if (iR[ij]%2== 0) //Если элемент чётный
    {
        iCh[in_Ch_El]= iR[ij];
        ++in_Ch_El;
    } //if
    else //Если элемент нечётный
    {
        iNCh[in_NCh_El]= iR[ij];
        ++in_NCh_El;
    } //else
} //for_ij
//Объединение массивов: после нечётных элементов в массив
//iNCh добавлены чётные элементы из массива iCh
for (int ij= 0; ij< in_Ch_El; ++ij)
    iNCh[in_NCh_El + ij]= iCh[ij];
//Вывод массива
for (int ij= 0; ij< m; ++ij)
    Tab->Cells[2][ij + 1]= iNCh[ij];

```

Остаток целочисленного деления отличен от нуля только для нечётных элементов массива, поэтому проверка на равенство ($iR[ij]\%2==0$) позволяет рассортировать элементы исходного массива iR по признаку *чётный-нечётный*.

При изучении методов сортировки недопустимо расточительным считалось применение двух переменных-массивов при сортировке лишь одного массива. Здесь же для обработки единственного массива привлекается сразу три переменные-массивы. Конечно, это очень незначительно и поэтому далее рассмотрим один из возможных алгоритмов, который при решении поставленной задачи кроме исходного массива iR никаких других переменных-массивов не требует.

6.4.2. Применение одного массива

Сделайте копию приложения *Линейная сортировка* (или *Сортировка методом пузырька*) с заголовком: *Перестановки нечётные-чётные*. Оставьте без изменения функции *FormCreate* и *Vvod_Dl_MasChange*. В функции *PyskClick* не изменяйте фрагменты, выполняющие ввод и вывод массива, а часть функции, в которой осуществлялась сортировка, замените новым блоком.

В этом блоке введите три простые переменные:

```
int iNachaln, iKonech, Byf;
```

Элементы исходного массива iR теперь рассматриваются не слева направо, как ранее, а с его обоих краёв. Переменная $iNachaln$ предназначена для записи текущего номера элемента массива при его просмотре с левого края (с начала), а переменная $iKonech$ – с его правого края (с конца). В начале анализа: $iNachaln=0$;

$iKonech = m - 1$. Напоминаем о том, что в m хранится число элементов фактически введенного массива, а зарезервированное число элементов массива на стадии компиляции программы определяется константой n ($n \geq m$). В ходе анализа текущие значения $iNachaln$ и $iKonech$ будут соответственно увеличиваться и уменьшаться.

Согласно условию задачи все элементы массива требуется упорядочить так, чтобы они образовывали две смежные группы из нечётных и чётных элементов: вначале в массиве должна располагаться группа нечётных элементов, а за ней – группа всех чётных элементов. Число чётных и нечётных элементов исходного массива заранее неизвестно, может оказаться, что все элементы нечётные либо только чётные. Поэтому *последовательно* будем исследовать на нечётность элементы, расположенные, например, в начале массива.

Рассмотрим нулевой элемент. Если этот элемент нечётный, то он стоит уже в нужной части массива, и поэтому менять его расположение совершенно не требуется, а следует сразу же перейти к анализу первого элемента (при этом $iNachaln = 1$). Если первый элемент также нечётный, то тогда следует заняться изучением чётности второго элемента (при этом $iNachaln = 2$) и так далее.

Может случиться так, что какой то из последовательно рассматриваемых элементов (с номером $iNachaln$ -ый) оказался чётным. Что с ним делать? Ведь и все последующие элементы могут быть чётными – тогда и предпринимать ничего не нужно: весь массив упорядочен по требованиям задачи. Но может реализоваться и такой вариант: среди оставшихся элементов встретится один или несколько нечётных элементов. Возможно даже, что все, кроме рассматриваемого элемента, будут нечётными.

Поэтому, разумнее всего, подыскать для размещения первого найденного чётного элемента ячейку в самом конце массива, где и должна располагаться группа из таких элементов. Наиболее пригодна для упомянутого размещения *последняя* ячейка массива с номером $iKonech = m - 1$. Если изначально в ней находился нечётный элемент, то содержимое найденной и последней ячеек поменяем местами и смело приступим к анализу следующего элемента ($iNachaln + 1$), и если он также чётный, поменяем его местами с предпоследним элементом при условии, что тот нечётный. Такой процесс можно продолжать далее.

Но не зря упоминалось о том, что обмен обитателями ячеек следует производить лишь в том случае, если они разной чётности. Если же последняя и предпоследняя ячейки исходного массива содержат чётные элементы, то такой обмен не имеет смысла: зачем же одно чётное значение ячейки заменять другим. В этом случае следует испытать третью с конца ячейку, если и она не подойдёт, то на предмет такого обмена посмотреть четвёртую, пятую и последующие ячейки. И только при обнаружении в ходе такого перебора ячеек ячейки, в которой находится нечётное значение – переместить его в $iNachaln$ -ую ячейку, а содержимое последней займёт ячейку с индексом $iKonech$.

По мере развития этих процессов левая и правая части массивов приводятся в порядок согласно упомянутому условию. При этом левая группа ячеек расши-

ряется вправо, а правая – влево. Ясно, что между ними ведётся *взаимновыгодное* сотрудничество, поэтому кампания наступления $iNachaln$ и кампания отступления $iKonech$ будут продолжаться лишь до нарушения условия $iNachaln < iKonech$. Поскольку при его нарушении массив будет полностью упорядоченным и поэтому строительство требуемого порядка следует завершить.

Сформулируем в более кратком виде описанный алгоритм сортировки.

1. Последовательно просматривать элементы массива iR , начиная с нулевого, до обнаружения чётного $iNachaln$ -го элемента.
2. Начать просматривать элементы массива с его конца и первый найденный нечётный $iKonech$ -ый элемент поменять местами с элементом $iNachaln$, обнаруженным на шаге 1.
3. Установить новые индексы для $iNachaln$ и $iKonech$ элементов: $iNachaln = iNachaln + 1$, $iKonech = iKonech - 1$.
4. Шаги 1, 2 и 3 продолжать до тех пор, пока выполняется условие $iNachaln < iKonech$.

С применением цикла *while* все шаги описанного алгоритма реализуются следующим фрагментом функции.

```
...
int iNachaln, iKonech; // Для хранения номера элемента //массива
int Byf; //Для временного хранения элемента массива
//Осуществляем сортировку
iNachaln= 0;
iKonech= m - 1;
while (iNachaln< iKonech)
{
    if (iR[iNachaln]%2!= 0)
        iNachaln++;
    else
        if (iR[iKonech]%2!= 0)
        {
            Byf= iR[iNachaln];
            iR[iNachaln]= iR[iKonech];
            iR[iKonech]= Byf;
            iNachaln++;
            iKonech--;
        }//if
    else
        iKonech--;
}// while
```

Успешно справится с заданной сортировкой и алгоритм, в котором вначале чётность элементов проверяется в конце массива. При прежней скорости работы ясность реализации такого алгоритма будет несколько ухудшена.

Небольшие изменения рассмотренного алгоритма позволяют:

- ☐ за четными элементами расположить нечетные элементы;
- ☐ за положительными элементами разместить отрицательные элементы (или наоборот);

- вначале сгруппировать все элементы, которые будут больше (или меньше) заданного числа, а за ними – все элементы меньше (или больше) этого числа.

6.4.3. Упорядочивание массива методом Ларионова

В 2004 г. четырнадцатилетний Матвей Ларионов на занятиях по изучению *TurboPascal* предложил свой очень оригинальный метод сортировки. Как и ранее сущность преобразования исходного массива состоит в том, чтобы вначале расположить все нечётные элементы, а за ними – все чётные. Не расстраивайтесь, если даже после нижеследующих пояснений, изучения кода этого метода и его проверки на компьютере вам не сразу станет ясным принцип его работы (логика мыслей гениев понятна далеко не всегда).

В этом методе при последовательном просмотре каждого элемента исходного массива iR с номером ij осуществляется подсчёт *текущего* числа ik нечётных элементов (он начинается с единицы). В начале просмотра $ik == 0$. В ходе анализа элементов массива может возникнуть две ситуации:

1. Все пересмотренные элементы являются нечётными, поэтому их расположение изменять не следует.
2. За (слева) *текущим* нечётным элементом находится один или больше чётных элементов. Поэтому упомянутый нечётный элемент и первый слева чётный элемент нужно поменять местами. В результате обмена правая граница группы ячеек с нечётными числами и левая граница совокупности ячеек с чётными числами продвигаются вправо. После анализа последнего элемента массива требуемый порядок будет установлен.

Для изучения и испытания этого алгоритма используйте предшествующее приложение, в котором фрагмент функции *PyskClick*, обеспечивающий упорядочивание массива, замените таким кодом:

```
//Осуществляем сортировку по Ларионову
int ik= 0;//Число нечётных элементов перед просмотром iR
for (int ij= 0, iByf; ij< m; ++ij)
    if (iR[ij]%2== 1)//Если число нечётное
    {
        iByf= iR[ik];//Временное хранение чётного элемента
        iR[ik]= iR[ij];//Меняем местами текущий нечётный элемент
        iR[ij]= iByf;// с первым чётным
        ik++;//Счётчик числа нечётных элементов
    }//if
```

Если в группе начальных ячеек (или во всём массиве) находятся нечётные числа, то содержимое ячеек $iR[ik]$ и $iR[ij]$ меняется само с собой, поскольку в этом случае $ij == ik$. В приведенном коде переменная ik фактически является номером ячейки, следующей за группой проанализированных ячеек с нечётными значениями. Если в этой ячейке и нескольких последующих находятся чётные числа (при этом $ij > ik$), то произойдёт обмен значениями очередной ij -ячейки с нечётным числом и ячейки с номером ik , в котором содержится крайнее слева чётное число.

6.5. Схемы алгоритмов

Составить программу решения сложной задачи, как правило, удаётся лишь с привлечением схем алгоритмов. Схема алгоритма – это совокупность графических блоков с указанием связей, существующих между ними. У одного и того же алгоритма может быть несколько взаимодополняющих друг друга схем. Может быть общая схема, которая лишь в основных чертах поясняет этапы его функционирования. Отдельные блоки этой схемы может детализировать целый набор вспомогательных схем, которые вносят ясность в процессы её функционирования. Блочно-модульный принцип положен в основу как схем всех электронных устройств, так и схем алгоритмов.

К схемам алгоритмов прибегают в следующих случаях:

- ☐ в процессе перехода от словесного описания алгоритма к написанию его кода на языке программирования;
- ☐ при разработке программ с очень сложной логикой работы;
- ☐ для изучения «чужих» программ с целью их модернизации или заимствования отдельных фрагментов;
- ☐ в сопроводительной документации к программе.

Рассмотрим основные компоненты схем алгоритмов.

Начало и конец программы отмечают овалом или скруглённым прямоугольником с соответствующими словами (см. рис. 6.11). Для этих же целей применяют и окружность, однако для размещения того же текста она требует больших вертикальных габаритов, что не всегда удобно.

Блок программы, в котором производится преобразование данных или вычисления по каким-либо формулам, изображается прямоугольником. Для изображений формул разрешается применять *любые* математические символы. Пример вычислительного блока показан на рис. 6.12.

Шаги сформулированного алгоритма выполняются последовательно один за другим. При этом блоки алгоритма, которые реализуют эти шаги, размещаются обычно один под другим и выполняются сверху вниз. Однако, когда прибегают к использованию операторов условий или выбора, схема алгоритма разветвляется. Ветвление алгоритма обозначают ромбом с выходящими из его боковых вершин горизонтальными отрезками прямой. Над одним отрезком помещают слово *Да*, а над другим – слово *Нет* (см. рис. 6.13).

Процессы, связанные с вводом и выводом данных, изображают параллелограммами, внутри которых указывается суть конкретной операции (см. рис. 6.14).

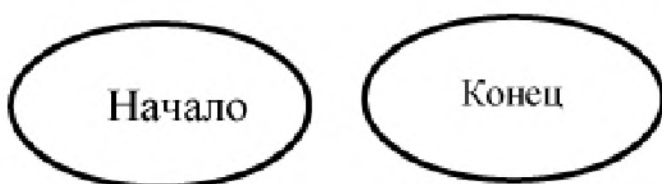


Рис. 6.11. Начальный и конечный элемент программы

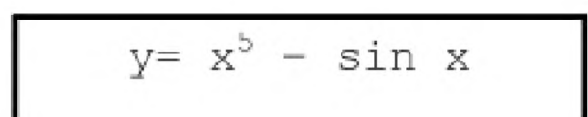


Рис. 6.12. Вычисления

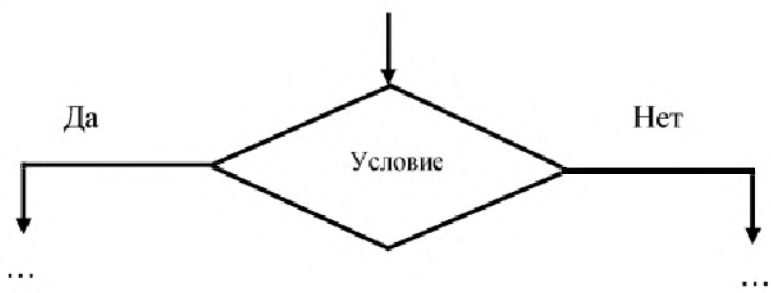


Рис. 6.13. Ветвление алгоритма

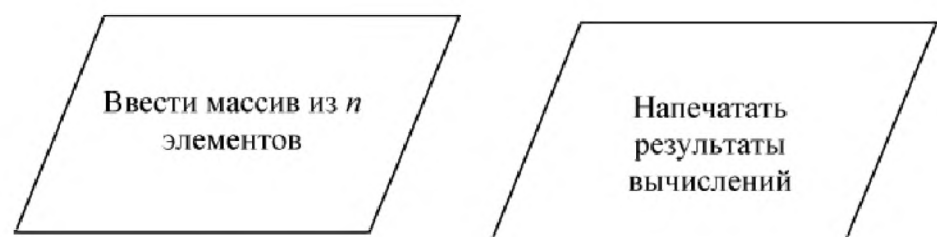


Рис. 6.14. Операции ввода-вывода

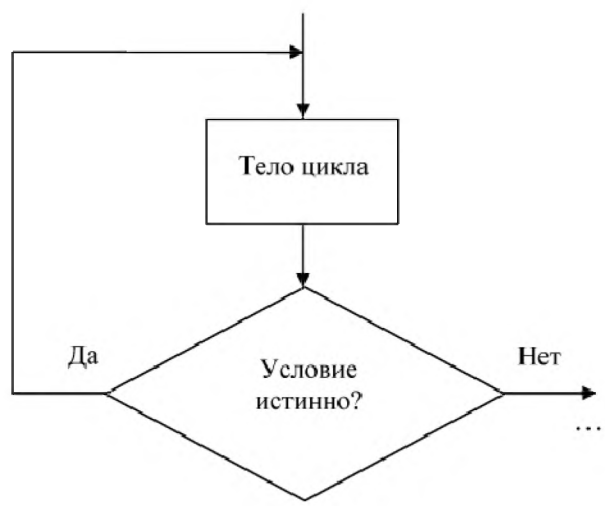


Рис. 6.15. Пример схемы для цикла *do-while*

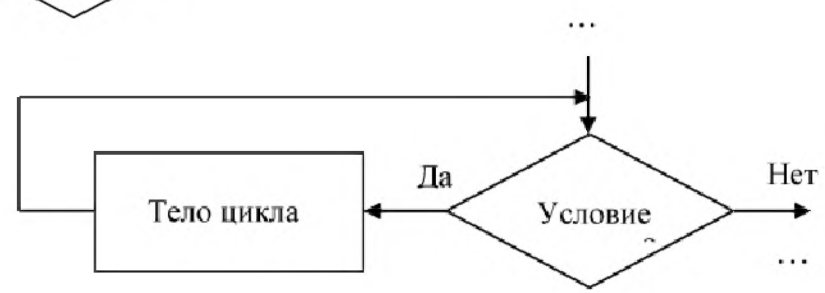


Рис. 6.16. Пример схемы для цикла *while*

Из указанных блоков можно монтировать алгоритмы любой сложности. В качестве примера на рис. 6.15 и рис. 6.16 иллюстрируются графические аналоги циклов *do_while* и *while*.

При разработке программ и их фрагментов широко используется метод, называемый методом разработки программ *сверху вниз*, его ещё именуют методом *пошаговой детализации*. Проиллюстрируем этот метод на примере задачи, рассмотренной в п. 6.4.2.

Напомним вначале словесное описание *основных* этапов ее решения.

1. Заполнить в таблице строк m ячеек целочисленными значениями.
2. Заполнить массив iR значениями, взятыми из ячеек таблицы строк.
3. Переставить элементы введенного массива таким образом, чтобы вначале располагалась группа из всех нечётных элементов, а после неё – группа из всех четных элементов.
4. Вывести в таблицу строк преобразованный массив чисел.

На рис. 6.17 показана схема основных операций алгоритма, составленная на основе приведенного словесного описания алгоритма.

Первый, второй и четвёртый шаги этого алгоритма многократно реализовывались нами в предшествующих программах, они достаточно просты и поэтому не требуют какой-либо детализации. Вместе с тем третий шаг алгоритма нуждается в пояснении, следует детализировать процесс, который приводит к требуемой перестановке элементов. Словесное описание алгоритма работы прямоу-

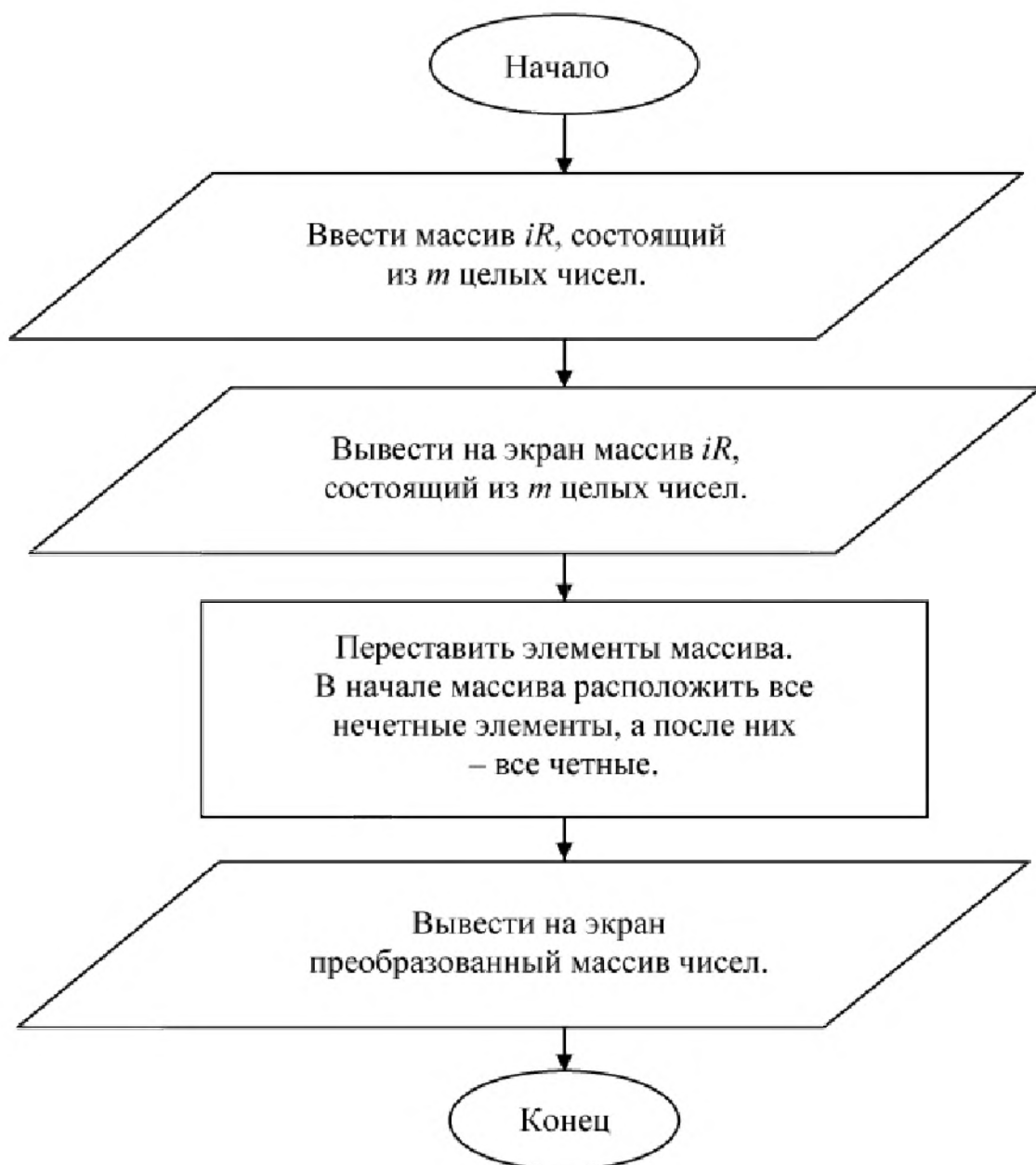


Рис. 6.17. Общая схема алгоритма

гольного блока приведено в п. 6.4.2. В этом алгоритме, в частности, отмечается, что перестановки элементов необходимо продолжать до тех пор, пока справедливо условие $iNachaln < iKonech$. Запуск на выполнение такого механизма (работа самого механизма пока не рассматривается) может осуществить как цикл *do_while*, так и цикл *while*. В п. 6.4.2. для этой же цели использовался цикл *while*. Поэтому для графической иллюстрации упомянутой части алгоритма воспользуемся этим же циклом. Таким образом, прямоугольник предшествующей схемы детализируется схемой, показанной на рис 6.18.

Схема работы прямоугольника, *тела* этого цикла, дана на рис. 6.19, она является последним шагом детализации рассмотренного алгоритма. Словесное описание алгоритма приведено в п. 6.4.2

Детализированная полная схема алгоритма в связи с её громоздкостью приводиться не будет. Такая схема иногда оказывается полезной в технической документации к программе. В большинстве же случаев метод пошаговой детализации достаточно наглядно иллюстрируется набором блок схем, представляющих собой разную глубину проработки отдельных частей полного алгоритма. Фактически такой набор схем показывает весь ход мыслей создателя программы от первых его замыслов к проработке отдельных элементов, и такая логика хорошо понятна специалистам, изучающим программу. Если же в инструкции к программе ограничиться *только* полной подробной схемой алгоритма, то это не в полной мере выполнит нагрузку, которая на неё возлагается.

Помимо проиллюстрированного метода разработки программы *сверху-вниз*, в практике широко используется также метод конструирования *снизу-вверх*. Он заключается в детальной проработке отдельных элементов полной задачи, а затем – в сборке из этих «кирпичей» всего «здания» программы. Вопрос, какой из этих методов предпочтительнее просто неуместен – это определяется опытом работы, пристрастиями, самой задачей, интуицией и вдохновением. Ведь програм-

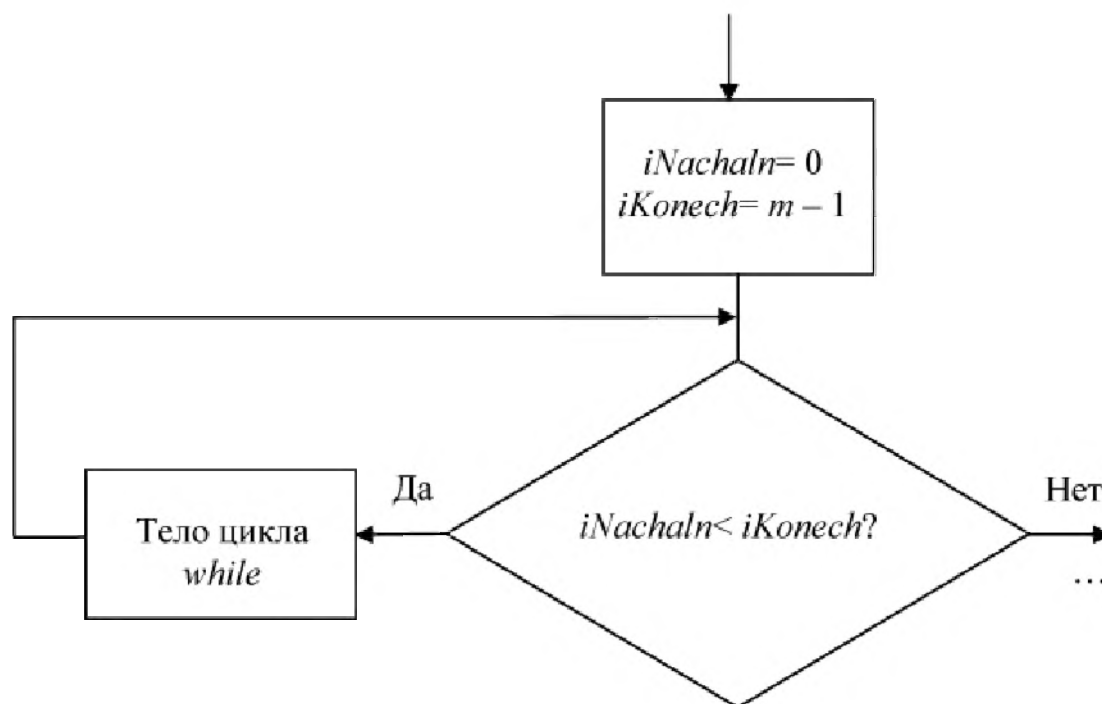


Рис. 6.18. Включение цикла *while* в общий алгоритм

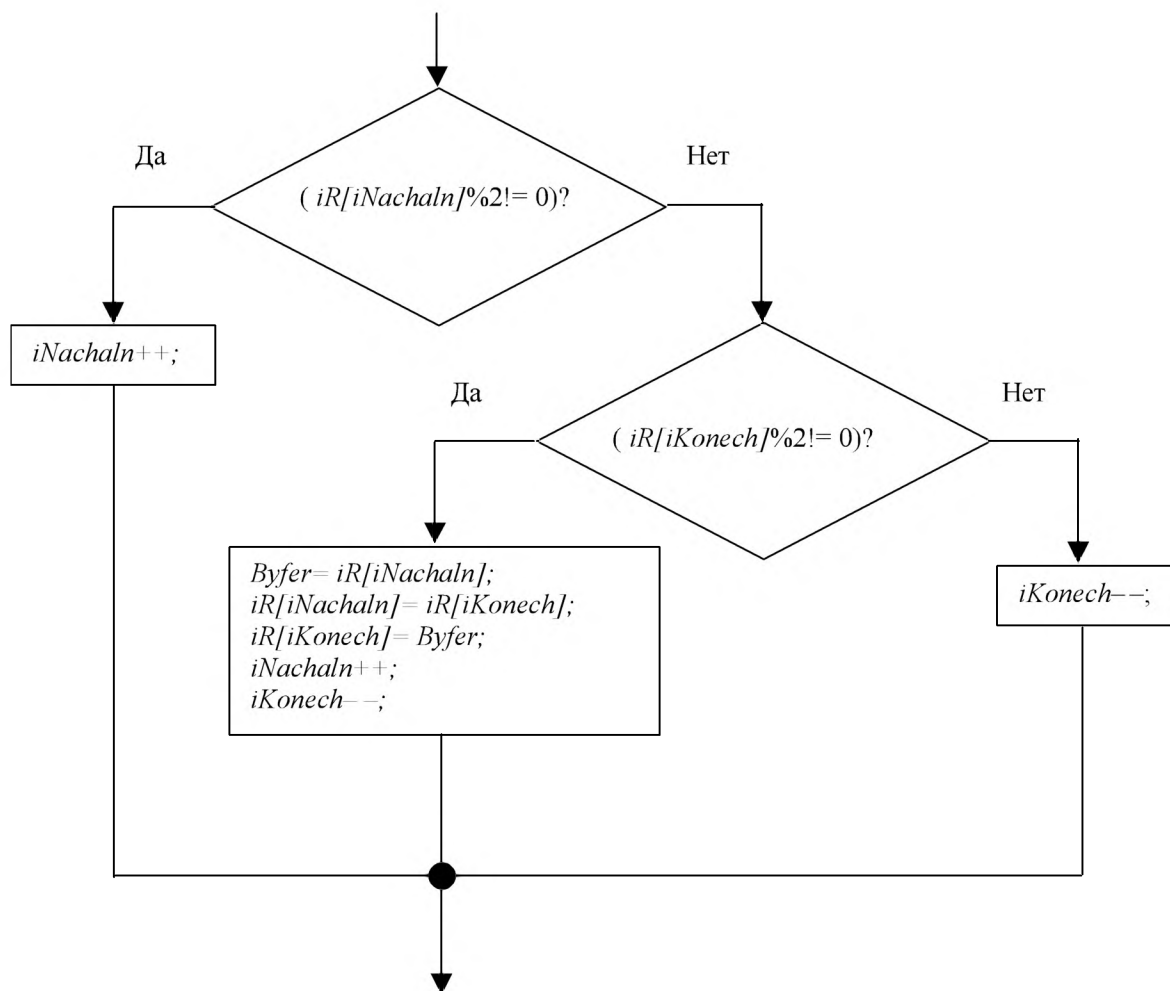


Рис. 6.19. Алгоритм работы прямоугольника (тела цикла *while*), приведенного на рис.6.18

мирование, как и всякий творческий процесс, сродни искусству. Поэтому вооружитесь обоими методами и либо применяйте их по отдельности, либо комбинируйте их использование при разработке одной и той же программы – это также бывает весьма плодотворно.

Вопросы для самоконтроля

- ☐ Зачем нужны массивы?
- ☐ Укажите индекс последнего элемента массива, состоящего *n* из ячеек.
- ☐ Проиллюстрируйте способ инициализации массива при его объявлении? Какое при этом существует ограничение?
- ☐ Какие массивы называют статическими?
- ☐ Как оценить максимальный размер глобального и локального статических массивов в конкретной программе?
- ☐ Назовите методы сортировки, рассмотренные на этом уроке. В чем суть их различий?
- ☐ В каких случаях применяются схемы алгоритмов?
- ☐ В чем заключается способы разработки программ *сверху-вниз* и *снизу-вверх*?

6.6. Задачи для программирования

Задача 6.1. Массив M , состоящий из 30 элементов, преформировать так, чтобы вначале стояли все положительные и равные нулю элементы в порядке убывания их значений, а затем все отрицательные в порядке возрастания значений.

Задача 6.2. Задан массив M , состоящий из n вещественных элементов. Определите суммы $S1$ и $S2$ положительных элементов массива соответственно с четными и нечетными номерами. Если $S1 > S2$, то поменяйте местами каждую соседнюю пару элементов массива (первый со вторым, третий с четвертым и т. д.). Если же $S1 \leq S2$, тогда поменяйте местами его элементы следующим образом: первый с последним, второй с предпоследним и т. д.

Задача 6.3. В упорядоченный по возрастанию значений массив M , состоящий из целых чисел, необходимо вставить число, не нарушив упорядоченности исходного массива.

Задача 6.4. Даны два массива $M1$ и $M2$, состоящие соответственно из $n1 = 8$ и $n2 = 11$ целочисленных элементов. В массив $M3$ отобрать те элементы массивов $M1$ и $M2$, которые имеются в каждом из них, а в массив $M4$ разместить элементы, которые находятся либо в $M1$, либо в $M2$.

6.7. Варианты решений задач

Решение задачи 6.1. Интерфейс приложения, функции *FormCreate* и *Vvod_Dl_MasChange* такие же, как п. 6.2.4. В файле реализации введена глобальная константа:

```
const int n = 100; //Максимальная длина массива
```

Тело функции *PyskClick* приводится ниже.

```
//Введенная длина массива
const int m = StrToInt(Vvod_Dl_Mas->Text);
int iNachaln, iKonech, //Номера элементов массива
    in_Otric; //Число отрицательных элементов массива
float fByf, //Для временного хранения элемента массива
    fM[n]; //Исходный массив чисел
bool Sort;
Tab->RowCount = m + 1;
//Заносим исходные данные в массив fM[n]
for (int ij = 0; ij < m; ++ij)
    fM[ij] = StrToFloat(Tab->Cells[1][ij + 1]);
//Осуществляем сортировку
iNachaln = 0;
iKonech = m - 1;
while (iNachaln < iKonech)
    if (fM[iNachaln] >= 0)
        iNachaln++;
    else
        if (fM[iKonech] >= 0)
        {
            fByf = fM[iNachaln];
```



```

        fM[iNachaln]= fM[iKonech];
        fM[iKonech]= fByf;
        iNachaln++;
        iKonech--;
    }//if
    else
        iKonech--;
    // Определение числа отрицательных элементов
in_Otric= 0;
for (int ij= 0; ij< m; ++ij)
    if (fM[ij]< 0)
        in_Otric++;
    //Линейная сортировка неотрицательных элементов массива
    // по убыванию их значений
for (int ij= 0; ij< m - in_Otric - 1; ++ij)
    for (int iCan= ij + 1; iCan< m - in_Otric; ++iCan)
        if (fM[ij]< fM[iCan])
        {
            fByf= fM[ij];
            fM[ij]= fM[iCan];
            fM[iCan]= fByf;
        }//if
    //Сортировка методом пузырька отрицательных элементов
    // массива по возрастанию их значений
do
{
    Sort= false;
    for (int ij= m - in_Otric; ij< m - 1; ++ij)
        if (fM[ij]> fM[ij + 1])
        {
            fByf= fM[ij];
            fM[ij]= fM[ij + 1];
            fM[ij + 1]= fByf;
            Sort= true;
        }//if
    }// do
while(Sort);

    //Вывод массива
for (int ij= 0; ij< m; ++ij)
    Tab->Cells[2][ij+1]= FloatToStrF(fM[ij], ffFixed, 8, 2);

```

Решение задачи 6.2. Приложение для этой задачи отличается от приложения по задаче 6.1 только телом функции *PyskClick*:

```

    //Введенная длина массива
const int m= StrToInt(Vvod_Dl_Mas->Text);
float fByf, //Для временного хранения элемента массива
        fM[n]; //Исходный массив чисел
Tab->RowCount= m + 1;
    //Заносим исходные данные в массив fM[n]
for (int ij= 0; ij< m; ++ij)
    fM[ij]= StrToFloat(Tab->Cells[1][ij + 1]);
int in_Ch; //Последний чётный индекс массива
    //Расчёт сумм fS1, fS2 положительных элементов массива

```

```

//соответственно с чётными и нечётными номерами
float fS1= 0, fS2= 0; //Задание и очистка переменных
for (int ij= 0; ij< m; ++ij)
    if (fM[ij]> 0)
        if ((ij + 1)%2== 0)
            fS1 += fM[ij]; //Сумма элементов с чётными номерами
        else
            fS2 += fM[ij]; //Сумма элементов с нечётными номерами
//Осуществляем перестановки элементов массива fM
if (fS1> fS2)
{
    //Вывод значения сумм fS1, fS2 и сообщения S1>S2
    ShowMessage("S1= " + FloatToStrF(fS1, ffFixed, 8, 2)+
        " S2= " + FloatToStrF(fS2, ffFixed, 8, 2)+ " : S1>S2");
    if (m%2== 0)
        in_Ch= m; //Если длина массива чётная
    else
        in_Ch= m - 1; //Если длина массива нечётная
    for (int ij= 0; ij<= in_Ch - 1; ++ij)
    {
        fByf= fM[ij];
        fM[ij]= fM[ij + 1];
        fM[ij + 1]= fByf;
        //Исключаем выход за верхнюю границу цикла
        if (ij< in_Ch - 1)
            ij++;
    } //for_ij
} //if (fS1>fS2)
else // (fS1<=fS2)
{
    //Вывод значения сумм iS1, iS2 и сообщения S1<= S2
    ShowMessage("S1= " + FloatToStrF(fS1, ffFixed, 8, 2)+
        " S2= " + FloatToStrF(fS2, ffFixed, 8, 2) + " : S1<= S2");
    //Цикл до середины массива длиной m
    for (int ij= 0; ij< m/2; ++ij)
    {
        fByf= fM[ij];
        fM[ij]= fM[m - ij - 1];
        fM[m - ij - 1]= fByf;
    } //for_ij
} //else (fS1<= fS2)
//Вывод массива
for (int ij= 0; ij< m; ++ij)
    Tab->Cells[2][ij+1]= FloatToStrF(fM[ij], ffFixed, 8, 2);

```

Решение задачи 6.3. Интерфейс, приведенный на рис. 6.8, дополните объектом типа *TEdit* с именем *Vvod_Chisla*, предназначенным для ввода вставляемого числа. Функции *FormCreate* и *Vvod_Dl_MasChange* такие же, как в приложениях по задачам 6.1 и 6.3, они описаны в п. 6.2.4. Тело функции *PyskClick* приведено ниже.

```

//Введенная длина массива
const int m= StrToInt(Vvod_Dl_Mas->Text);
//Ввод вставляемого числа
const float fChislo= StrToFloat(Vvod_Chisla->Text);

```

```
const int n = 100; //Максимальная длина всех массивов
int n1 = 8, n2 = 11;

void __fastcall TZadacha6_4::FormCreate(TObject *Sender)
{
    n1 = StrToInt(Vvod n1->Text);
```

```

n2= StrToInt(Vvod_n2->Text);
if (n1> n2)
    Tab->RowCount= n1 + 1;
else
    Tab->RowCount= n2 + 1;
//Называем столбцы таблицы
Tab->Cells[0][0]= "№";
Tab->Cells[1][0]= "M1";
Tab->Cells[2][0]= "M2";
Tab->Cells[3][0]= "M3";
Tab->Cells[4][0]= "M4";
//Нумерация всех строк
for (int ij=0; ij< n; ++ij)
    Tab->Cells[0][ij+1]= ij + 1;
} //FormCreate

void __fastcall TZadacha6_4::Vvod_n1Change(TObject *Sender)
{
    n1= StrToInt(Vvod_n1->Text);
    if (n1>n2)
        Tab->RowCount= n1 + 1;
    else
        Tab->RowCount= n2 + 1;
} //Vvod_D1_MasChange

void __fastcall TZadacha6_4::Vvod_n2Change(TObject *Sender)
{
    n2= StrToInt(Vvod_n2->Text);
    if (n1>n2)
        Tab->RowCount= n1 + 1;
    else
        Tab->RowCount= n2 + 1;
} //Vvod_n2Change

void __fastcall TZadacha6_4::PyskClick(TObject *Sender)
{
    //Введенная длина массива
    n1= StrToInt(Vvod_n1->Text),
    n2= StrToInt(Vvod_n2->Text);
    int iM1[n], iM2[n], //Исходные массивы целых чисел
        iM3[2*n], iM4[2*n], iSymma_M1_M2[2*n];
    if (n1>n2)
        Tab->RowCount= n1 + 1;
    else
        Tab->RowCount= n2 + 1;
    //Заносим исходные данные
    for (int ij= 0; ij< n1; ++ij)
        iM1[ij]= StrToFloat(Tab->Cells[1][ij + 1]);
    for (int ij= 0; ij< n2; ++ij)
        iM2[ij]= StrToFloat(Tab->Cells[2][ij + 1]);
    //Формирование массива M3
    //Очистка счётчика in3 для определения длины массива M3
    int in3= 0;
    for (int ii= 0; ii< n1; ++ii)
        for (int ij= 0; ij< n2; ++ij)
            if (iM1[ii]== iM2[ij])

```

```

        {
            iM3[in3]= iM1[ii];
            in3++;
        }// if
//Объединение массивов iM1 и iM2 в один массив iSymma_M1_M2
for (int ij= 0; ij< (n1 + n2); ++ij)
    if (ij< n1)
        iSymma_M1_M2[ij]= iM1[ij];
    else
        iSymma_M1_M2[ij]= iM2[ij - n1];
//Формирование массива M4
//Очистка счётчика in4 для определения длины массива M4
int in4= 0;
bool bx;
for (int ii= 0; ii< (n1 + n2); ++ii)
{
    bx= false;
    for (int ij= 0; ij< in3; ++ij)
        if (iM3[ij]== iSymma_M1_M2[ii])
            bx= true;
    if (bx== false)
    {
        iM4[in4]= iSymma_M1_M2[ii];
        in4++;
    }//if
}//for_ii
//Вывод массивов
for (int ij= 0; ij< in3; ++ij)
    Tab->Cells[3][ij + 1]= iM3[ij];
for (int ij= 0; ij< in4; ++ij)
    Tab->Cells[4][ij + 1]= iM4[ij];
}//PyskClick

```

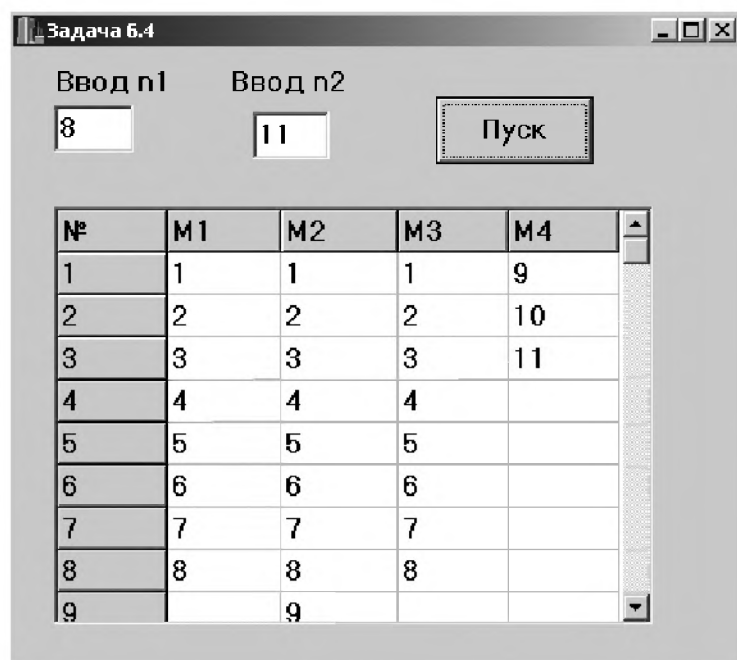


Рис. 6.20. Интерфейс приложения Задача 6.4

Урок 7. Многомерные массивы

7.1. Примеры многомерных массивов

В табл. 7.1 показаны фрагменты ведомости успеваемости 10-А класса за первую четверть учебного года. Оценки в ней выставлены по двенадцатибальной системе.

Таблица 7.1. Ведомость успеваемости 10-А класса за первую четверть учебного года

Номер по списку	Фамилия	Математика	Физика	Английский	Информатика	Физкультура
1	Иванов	11	9	8	12	10
2	Петров	7	5	2	9	12
3	Сидоров	5	7	2	10	5
...						
n	Яковлев	12	12	12	12	12

Рассмотрим только *оценки* данной ведомости. Они представлены в виде *n строк* (число учащихся) и *5 столбцов* (число предметов). Эта таблица чисел – пример *двумерного* массива или *матрицы*.

Данные, приведенные в табличном виде, удобно просматривать, корректировать и анализировать. Проиллюстрируем процесс обработки данных более подробно. Например, для анализа успеваемости класса можно вычислить следующие характеристики:

- ☐ среднюю успеваемость класса – среднее арифметическое значение всех оценок;
- ☐ среднюю успеваемость каждого ученика;
- ☐ среднюю успеваемость класса по каждому предмету.

Такие и аналогичные вычисления являются примерами различной обработки табличных данных или двумерных массивов.

Ведомости успеваемости класса за *другие* четверти года также представляют собой аналогичные двумерные массивы. А вот данные всех *четырёх* четвертей в совокупности уже характеризуют успеваемость класса за *весь* учебный год и являются примером *трёхмерного* массива, *трехмерной* матрицы. Чтобы абстрагировано представить такую трехмерную структуру, каждую оценку всех ведомостей изобразим в виде сплошного кружка (см. рис. 7.1.).

Обработка упомянутой трёхмерной матрицы позволяет осветить успеваемость

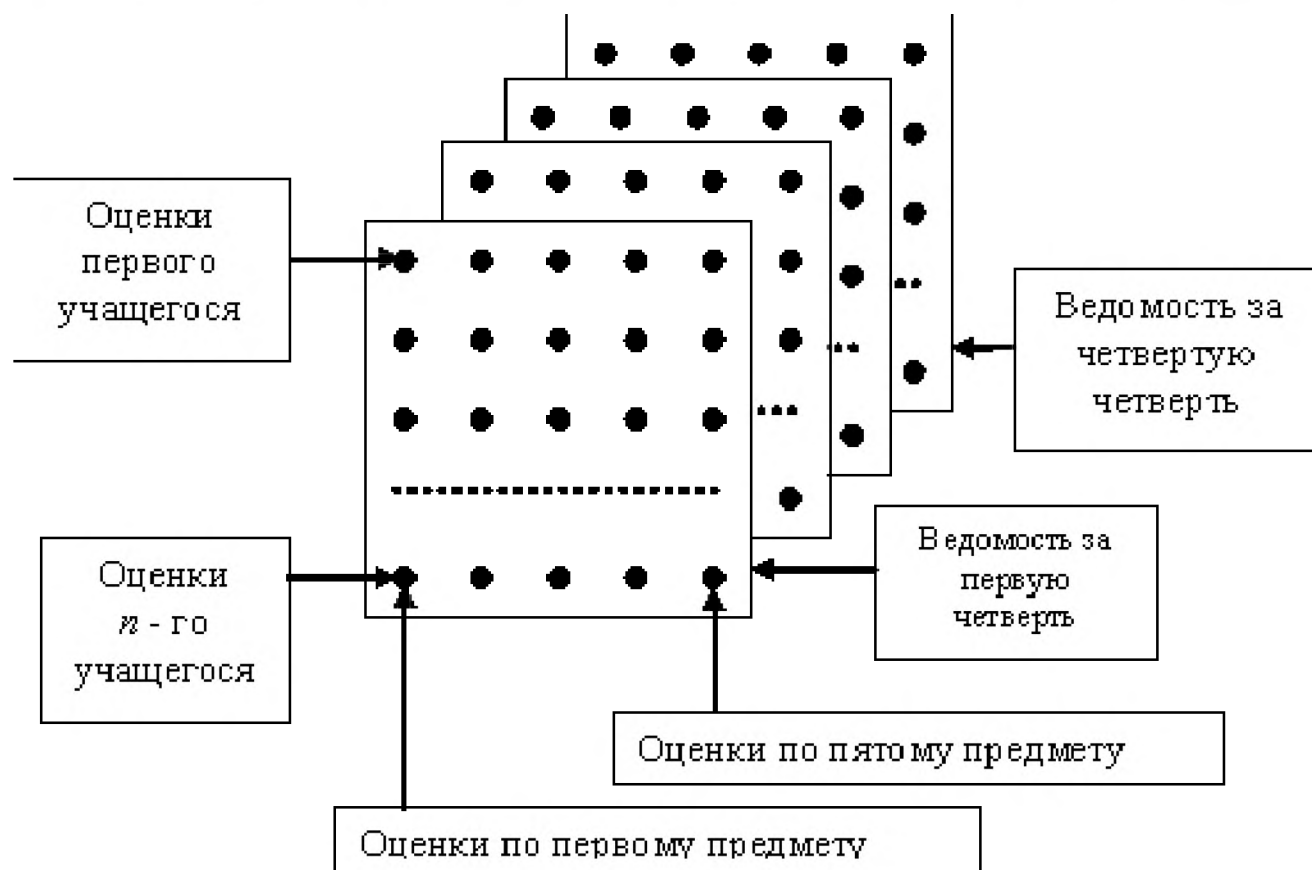


Рис. 7.1. Трехмерный массив

мость класса в масштабах всего года: определить среднюю успеваемость группы за весь учебный год, рассортировать учащихся по убыванию их среднегодовой успеваемости и многое другое.

7.2. Описание многомерных массивов

Описание многомерных массивов в программах проиллюстрируем на массивах, рассмотренных в предшествующем разделе.

```
const n= 23; // Число учеников в классе
//Двумерные массивы чисел типа int из n строк и 5 столбцов
int iChetv_1[n][5],
    iChetv_2[n][5], iChetv_3[n][5], iChetv_4[n][5];
int iVesj_God[n][5][4]; //Трехмерный массив чисел типа int
// n – число учащихся, 5 – число предметов,
// 4 – число четвертей года
```

В многомерных массивах число элементов массива по каждой размерности указывается в *квадратных* скобках, стоящих сразу же за именем массива. В двумерных массивах в таких скобках вначале приводится *число строк* (количество учащихся), а затем – *число столбцов* (количество предметов). В трехмерных массивах в отдельных квадратных скобках указывается ещё и число индексов по третьему измерению (число четвертей года). Между квадратными скобками допустимы пробелы (один и более), однако они лишь ухудшают восприятие переменной (её запись становится длиннее) и поэтому нецелесообразны.

Иногда массивы удобно определять с использованием зарезервированного слова *typedef* (*type* – тип, *define* – определять), применяемого при объявлении любых пользовательских типов данных или создании синонимов для любых уже существующих типов. Покажем этот способ при объявлении прежних массивов.

```
//Тип – двумерная матрица типа int
typedef int iChetvertj[n][5];
//Тип – трёхмерная матрица типа int
typedef int iGod[n][5][4];
//Двумерные массив чисел типа int
iChetvertj iChetv_1, iChetv_2, iChetv_3, iChetv_4;
//Трёхмерный массив чисел типа int
iGod iVesj_God;
```

В ряде случаев в целях наглядности кода удобно использовать менее компактную форму описания массивов. Проиллюстрируем её также на прежних примерах.

```
typedef int iPredmetu[5]; //Тип – одномерный массив (строка)
iPredmetu iChetv_1[n]; //Двумерный массив, массив массивов
typedef iPredmetu iYcheniki[n]; //Тип – двумерный массив
iYcheniki iVesj_God[4]; //Трёхмерный массив
```

Каждый из n элементов массива *iChetv_1* состоит из элементов типа *iPredmetu*, любой из них в свою очередь содержит по 5 элементов типа *int*. Поэтому массив *iChetv_1* (как и ранее) является двумерным. Каждый из четырёх элементов массива *iVesj_God* состоит из компонентов двумерного массива. При этом упомянутый двумерный массив представлен в виде *массива массивов*: столбец (или строка) из n компонент, представленных в виде одномерных массивов из 5 элементов типа *int*. Таким образом, *iVesj_God* – трёхмерный массив, выраженный через одномерные массивы. Все приведенные формы описания массивов полностью эквивалентны.

Существуют массивы *больших* размерностей, они описываются по прежним правилам, однако на практике встречаются исключительно редко. Число измерений массива ограничивается только *суммарным объёмом* памяти всех ячеек массива, который не должен превышать объём доступной памяти.

Рассмотрение многомерных массивов – хороший повод ещё раз напомнить общее определение (см. шестой урок): массивы – это тип данных, состоящий из *фиксированного* числа *однотипных* компонент, максимальное количество которых определяется типом этих компонент (и объёмом доступной памяти).

В частности, максимальное число учащих в переменной *iVesj_God* при её размещении в стеке объёмом 1 МБ будет составлять:

$n_Lokaln = Razmer_Steka / (4 \cdot 5 \cdot 4) = 1\,048\,576 / 80 = 13\,107$. Здесь 4 – количество четвертей, 5 – число предметов, 4 – размер типа.

В глобальном массиве это число оценивается по формуле:

$n_Globaln = Razmer_Svobodn_Operat_Pam / (4 \cdot 5 \cdot 4)$.

При оперативной памяти, например, в 256 МБ максимальное количество ячеек глобального массива в 256 раз больше, чем в локальном массиве.

7.3. Доступ к элементам массива

Способы доступа к элементам одномерных и многомерных массивов *полностью* аналогичны. Согласно ведомости успеваемости (см. табл. 7.1) учащийся под номером 2 (Петров) по третьему предмету (английский) получил оценку 2. Такая оценка по двенадцатибалльной шкале не очень украшает Петрова, но делает запоминающимся пример по присваиванию указанного значения соответствующему элементу двумерного массива:

```
iChetv_1[1][2] = 2;
```

Ещё раз взгляните в табл. 7.1. Вы увидите, что на пересечении *второй* строки и *третьего* столбца матрицы-ведомости оценок стоит 2. Поскольку *Builder* начинает счёт с нуля, поэтому номера строки и столбца уменьшены на единицу. Таким образом, табличное значение в память компьютера переведено совершенно правильно. Согласно сформулированному правилу, в первой квадратной скобке указан номер строки, а затем, во второй квадратной скобке, – номер столбца элемента массива, к которому осуществляется доступ.

Вот как выглядит операция присваивания конкретного значения «3» элементу (учащемуся), расположенному на пересечении строки с номером *ii* и столбца с индексом *ij*:

```
iChetv1[ii][ij] = 3;
```

Вопросы для текущего самоконтроля

- ☐ Правомерен ли такой оператор присваивания?

```
iChetv1[ii][ij] = 4.5;
```

- ☐ Какое число храниться в ячейке *Chetv1[ii][ij]* после выполнения оператора присваивания?

Ниже приводится два способа инициализации одними и теми же значениями двумерного массива *iMass_2*, состоящего из *трёх* строк и *двух* столбцов.

```
int iMass_2[3][2] = {{1,1}, {0,2}, {1,0}};  
int iMass_2[3][2] = {1, 1, 0, 2, 1, 0};
```

Таким образом, при инициализации многомерного массива он представляется либо как массив из массивов, либо задаётся общий список элементов в том порядке, в каком элементы расположены. В первом случае каждый массив заключается в свои фигурные скобки. В каждом из этих способов *разрешается* не указывать последнюю размерность при описании массива:

```
int iMass_2[3][] = {{1,1}, {0,2}, {1,0}};  
int iMass_2[3][] = {1, 1, 0, 2, 1, 0};
```

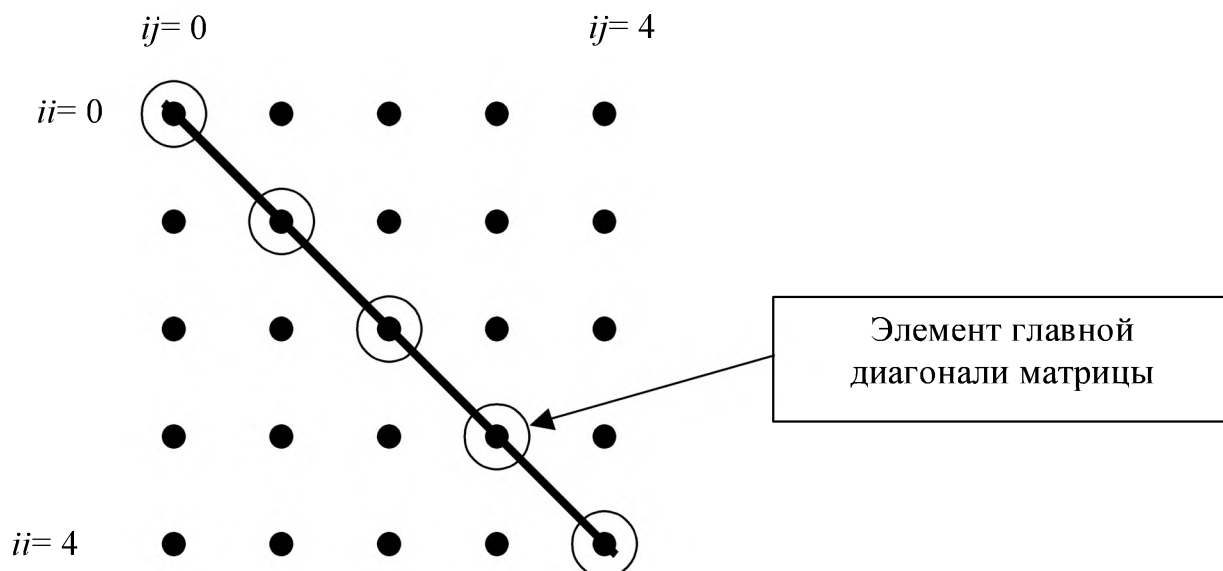


Рис. 7.2. Главная диагональ матрицы

7.4. Главная и побочная диагонали

Если число строк матрицы равно числу столбцов, то такая матрица называется квадратной и имеет *главную* и *побочную* диагонали. Схематично квадратная матрица показана на рис. 7.2. Все её элементы изображены в виде черных кружков. Но среди этих элементов легко заметны и сияющие улыбкой счастливчики: они отмечены полыми кружками и соединены отрезками прямых. Да, эти отмеченные нимбом гордецы принадлежат к главной диагонали! Элементам матрицы, расположенным на отрезке прямой, который соединяет левый верхний и нижний правый углы матрицы, очень повезло – они расположены на главной диагонали.

У всех элементов главной диагонали, у этих баловней судьбы, есть чем гордиться. В когорту представителей голубой крови их выводит удачное расположение: номер строки ii совпадает с номером столбца ij . Таким образом, условием нахождения элемента матрицы на её главной диагонали, является выполнение равенства $ii = ij$. При $ii > ij$ элементы находятся под главной диагональю, а при $ii < ij$ – над ней (см. рис. 7.2).

Побочной диагонали принадлежат элементы, через которые проходит отрезок прямой, соединяющий верхний правый и левый нижний углы квадратной матрицы. Эти элементы на рис. 7.3 отмечены нимбами, как и их собратья из главной диагонали. Ведь они относятся ко второй по важности диагонали квадратной матрицы. После просмотра упомянутой иллюстрации легко представить пространственное положение элементов на побочной диагонали. Однако обработку матриц должен производить компьютер, поэтому определим для него координаты расположения элементов побочной диагонали.

С этой целью рассмотрим матрицу, у которой число строк и столбцов равно n . Для такой матрицы элементы её побочной диагонали будут иметь следующие координаты (см. рис. 7.3):

$$ii = 0, \quad ij = n - 1$$

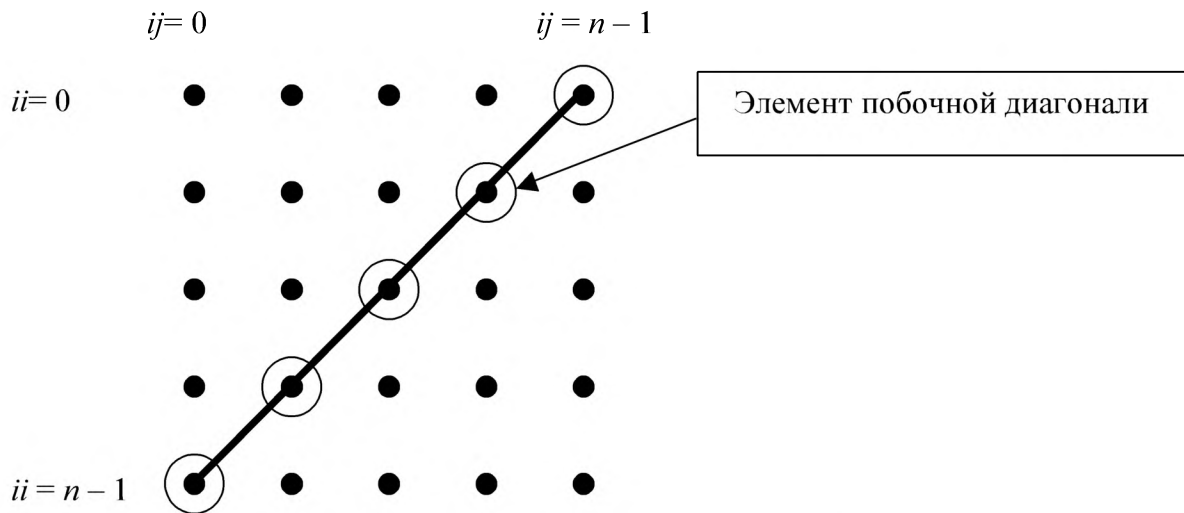


Рис. 7.3. Побочная диагональ матрицы

```

ii = 1,   ij = n - 2
ii = 2,   ij = n - 3
...
ii = n - 1,   ij = 0

```

Если в любой строке сложить левые и правые части уравнений для ii и ij , то в результате получим уравнение: $ii+ij=n-1$. Из него определяется следующее условие нахождения элемента квадратной матрицы на побочной диагонали: $ii=n-ij-1$. Эта формула связывает номер строки с номером столбца любого счастливого элемента побочной диагонали. Сделайте трассировку этой формулы при помощи приведенных выше координат, и убедитесь, что она верна.

Используя эту зависимость и рис. 7.3 нетрудно определить те элементы матрицы, что лежат под ($ii > n - ij - 1$) и над ($ii < n - ij - 1$) побочной диагональю. Приведенные неравенства понадобятся при решении задач.

7.5. Примеры обработки матриц

Обрабатывать матрицы научимся вначале на простых примерах. Пусть требуется найти максимальный элемент матрицы M и его координаты: номер строки и столбца, на пересечении которых он находится. Алгоритм нахождения максимума массива и все другие приёмы, необходимые для решения этой задачи, рассмотрены на прошлом уроке. Поэтому предлагаем *самостоятельно* проанализировать работу приложения *Использование матриц*, его интерфейс показан на рис. 7.4, функции *FormCreate*, *Chislo_StrokChange*, *Chislo_StolbcovChange* и *PyskClick* приведены ниже. В файле реализации введены глобальные константы:

```

const int n_St= 20, //Максимальное число строк
          n_Cl= 20; //Максимальное число столбцов

```

Тело функции *FormCreate*:

```

// Называем все строки таблицы. Их число равно n_St -
//максимальному числу строк матрицы
for (int ii= 1; ii<= n_St; ++ii)

```

Рис. 7.4. Интерфейс программы *Использование матриц*

```

    Tab->Cells[0][ii]=" стр_" + IntToStr(ii);
//Называем все столбцы таблицы. Их число равно n_Cl -
//максимальному числу столбцов матрицы
for (int ij= 1;ij<= n_Cl; ++ij)
    Tab->Cells[ij][0]="кол_" + IntToStr(ij);

```

Тело функции *Chislo_StrokChange*:

```

//Установка числа строк таблицы Tab
const int nSt= StrToInt(Chislo_Strok->Text);
Tab->RowCount= nSt + 1;

```

Тело функции *Chislo_StolbcovChange*:

```

//Установка числа столбцов таблицы Tab
const int nCl= StrToInt(Chislo_Stolbcov->Text);
Tab->ColCount= nCl + 1;

```

Тело функции *PyskClick*:

```

//Введенное число строк
const int nSt= StrToInt(Chislo_Strok->Text),
//Введенное число столбцов
    nCl= StrToInt(Chislo_Stolbcov->Text);
//Определение матрицы из n_St строк и n_Cl столбцов
float fM[n_St][n_Cl],
    fMax; // Максимальное значение элемента матрицы
int iSt_Max, iCl_Max; // Положение fMax
//Заносим исходные данные в массив fM[nSt][nCl]
for (int ii= 1; ii<= nSt; ++ii)
    for (int ij= 1; ij<= nCl; ++ij)
        fM[ii - 1][ij - 1]= StrToFloat(Tab->Cells[ij][ii]);
//Определяем максимальное значение элемента fMax
// и его местоположение iSt_Max, iCl_Max
fMax= fM[0][0];
iSt_Max= 0;
iCl_Max= 0;
for (int ii= 0; ii< nSt; ++ii)

```

```

for (int ij= 0; ij< nCl; ++ij)
    if (fM[ii][ij]> fMax)
    {
        fMax= fM[ii][ij];
        iSt_Max= ii;
        iCl_Max= ij;
    }//if
//Вывод результатов в объект типа TLabel с именем Rez,
//в его свойстве WordWrap следует установить true
Rez->Caption = "Максимальное значение = " +
FloatToStrF(fMax, ffFixed, 10, 2) + ". Оно находится на\ пересечении " +
IntToStr(iSt_Max + 1) + " строки» + " и " +
IntToStr(iCl_Max + 1)+ " столбца.";

```

Константы n_St и n_Cl задают *максимальное* число строк и столбцов матрицы. Они описаны *не внутри* отдельной пользовательской функции, а в файле реализации, поэтому являются *глобальными* (видимыми) для всех пользовательских функций этого файла (описанных после упомянутых констант). Часть файла, расположенная ниже обсуждаемых констант, играет роль логического блока их видимости.

В функции *PyskClick* упомянутые глобальные константы применяются для задания *локального* массива: `float fM[n_St][n_Cl]`. В теле этой же функции её *локальные* константы с похожими именами (nSt и nCl) получают *текущие* значения, введенные в полях ввода *Chislo_Strok* и *Chislo_Stolbcov*. Эти константы *не могут* участвовать в задании массива. Но, почему же не могут? – возразят некоторые читатели. Ведь на прошлом уроке отмечалось, что длина массива может задаваться либо константой, либо константным выражением, а nSt и nCl являются константами.

Давайте разберёмся с этим вопросом подробнее. Как известно, память для *статических* массивов выделяется на стадии компиляции, поэтому длину статического массива *нельзя* изменять в ходе работы программы. Если же для задания массива использовать константы nSt и nCl , то *Builder* в ходе компиляции выведет сообщение об ошибке: *длина массива должна задаваться либо константой, либо константным выражением*. В случае использования констант nSt и nCl для задания массива *Builder* не может выделить какой-либо объём памяти для размещения массива, поскольку на этапе компиляции программы ещё не известны значения, которые пользователь введёт в упомянутые константы (при помощи полей ввода) в ходе работы приложения.

Глобальные константы n_St и n_Cl применяются также в функции *FormCreate* (циклах *for*) для нумерации *всех* строк и столбцов объекта типа *TSringGrid*, который в нашем проекте называется *Tab*.

Теперь рассмотрим более сложный пример: требуется *в каждой* строке *прежней* матрицы fM найти наименьший элемент $fMin$ и все такие элементы записать в одномерный массив fR . Поскольку упомянутые операции иллюстрируются на основе прежнего приложения, поэтому ограничимся только показом дополнительных блоков функции *PyskClick*, решающих эти задачи.

```

Tab->ColCount= nCl + 2;//Дополнительный столбец в Tab
//В каждой строке матрицы fM находится наименьший элемент
//строки fMin и все такие элементы записываются в одномерный
//массив fR

//Определение fMin и массива fR из n_St элементов
float fMin, fR[n_St];
for (int ii= 0; ii< nSt; ++ii)
{
    fMin= fM[ii][0];
    for (int ij= 1; ij< nCl; ++ij)
        if (fM[ii][ij]< fMin)
            fMin= fM[ii][ij];
    fR[ii]= fMin;
} //for_ii
//Вывод результатов
for (int ii= 0; ii< nSt; ++ii)
    Tab->Cells[nCl+1][ii+1]= FloatToStrF(fR[ii] ,ffFixed, 10, 2);

```

Для вывода результирующего массива fR минимальных значений строк матрицы fM увеличено число столбцов объекта Tab на один столбец:

```
Tab->ColCount= nCl + 2;
```

Поиск минимального элемента *каждой* строки осуществляется при помощи алгоритма, подробно рассмотренного в п. 6.1.4. Такой деятельностью занимается вложенный цикл *for* по ij . Надеемся, вы уже вспомнили, почему начальный параметр этого цикла равен 1, а не 0.

Внешний цикл *for* по ii определяет *номер* строки, в которой выполняется упомянутый поиск. При обработке выбранной ii -ой строки перед циклом по ij предлагается «затравочный» кандидат на искомое минимальное значение $fMin$, затем, при содействии цикла по ij , определяется *истинный* минимальный элемент строки, после чего его значение записывается в соответствующей ячейке массива fR . На следующем шаге цикла найденное значение $fMin$ обработанной строки заменяется новым «затравочным» кандидатом для последующей строки.

На прошлом уроке изучены алгоритмы упорядочивания одномерных массивов. Строки, столбцы и диагонали матриц представляют собой одномерные массивы и поэтому их также можно подвергать сортировкам прежними методами.

Следующий блок программы на примере прежней матрице fM расставляет по возрастанию элементы столбца с номером n_Sort . Для его работы в предшествующем фрагменте оставьте только строку:

```
Tab->ColCount= nCl + 2;//Дополнительный столбец в Tab
```

Блоки, ответственные за сортировку и вывод столбца после его сортировки, приведены ниже.

```

//Сортировка столбца n_Sort по возрастанию
//элементов в матрице fM
int n_Sort = 2;
for (int ii= 0; ii< nSt - 1; ++ii)
    for (int iCan= ii + 1; iCan< nSt; ++iCan)
    {

```

```

    if (fM[ii][n_Sort - 1] > fM[iCan][n_Sort - 1])
    {
        float fx = fM[ii][n_Sort - 1];
        fM[ii][n_Sort - 1] = fM[iCan][n_Sort - 1];
        fM[iCan][n_Sort - 1] = fx;
    } // if
} // for_ii
// Вывод результатов
for (int ii = 0; ii < nSt; ++ii)
    Tab->Cells[nCl + 1][ii + 1] =
        FloatToStrF(fM[ii][n_Sort - 1], ffFixed, 10, 2);

```

После нажатия командной кнопки в результате работы нового варианта функции *PyskClick* произойдёт расширение таблицы строк на один столбец, в котором появятся элементы второго столбца исходной таблицы, отсортированные по возрастанию элементов.

Обогатим теперь учебную программу дополнительным заданием. В каждом столбце матрицы fM вычислим сумму квадратов fS её элементов, а затем среди всех этих сумм определим минимальную сумму $fMin$ и номер столбца iN , в котором она находится. В исходном приложении оставьте только ввод числа строк, столбцов и матрицы fM . Следующий фрагмент решает поставленное задание.

```

float fS[n_Cl], fMin;
int iN;
for (int ij = 0; ij < nCl; ++ij)
{
    float f_S = 0; // Сумма квадратов элементов текущего столбца
    for (int ii = 0; ii < nSt; ++ii)
        f_S += pow(fM[ii][ij], 2);
    fS[ij] = f_S;
    // Определение минимальной суммы и номера её столбца
    if (ij == 0)
    {
        fMin = fS[0];
        iN = 0;
    } // if_1
    if (fS[ij] < fMin)
    {
        fMin = fS[ij];
        iN = ij;
    } // if_2
} // for_ij
ShowMessage(" Значение минимальной суммы: " +
    FloatToStrF(fMin, ffFixed, 10, 2) + " находится в столбце " +
    IntToStr(iN + 1));

```

Алгоритм нахождения произведения элементов одномерных массивов применим, например, для поиска произведения fP элементов главной диагонали матрицы fM . Для его иллюстрации в исходном приложении в полях ввода числа задаваемых строк и столбцов матрицы введите одни и те же числа, поскольку диагонали имеются только у квадратных матриц. Напоминаем условие нахождения элемента на главной диагонали: $ii = ij$.

```
// Предварительная операция перед вычислением произведения
float fP= 1.0;
for (int ii= 0; ii< nSt; ++ii)
    for (int ij= 0; ij<nCl; ++ij)
        if (ii==ij)
            fP *= fM[ii][ij]; // или fP= fP*fM[ii][ij];
ShowMessage("Произведение элементов главной диагонали \
равно: " + FloatToStrF(fP, ffFixed, 10, 2));
```

После объявления *локальной* числовой переменной fP в ней хранится неопределённое значение. Вследствие этого, если опустить процесс инициализации в первой строке приведенного кода ($float fP= 1.0;$), то искомое произведение *всегда* будет неверным. При вычислении произведения элементов массива присваивание единичного значения переменной, предназначенной для хранения этого произведения, является *необходимой* операцией. Эта операция несколько аналогична очистке переменной $fSredn_Ocenka$ перед суммированием членов числового ряда (см. п. 6.1.3).

В обсуждаемом фрагменте выбор элементов главной диагонали производится при помощи вложенных циклов по ii и ij с применением условия $ii==ij$. Однако перемещаться по элементам главной диагонали можно и с использованием лишь одного цикла:

```
float fP= 1.0;
for (int ii= 0; ii< nSt; ++ii)
    fP *= fM[ii][ii];
```

Аналогичным способом можно организовать перемещение и по элементам побочной диагонали. Для этого используется условие нахождения элемента на побочной диагонали: $ii = n - ij - 1$. Ниже показано, как определить минимальный элемент побочной диагонали матрицы fM и его координаты:

```
float fMin= fM[0][nCl - 1]; //Минимальное значение
int iSt_Min= 0, //Его
    iCl_Min= nCl - 1; //местоположение
for (int ij= 0; ij< nCl; ++ij)
    if (fM[nCl - ij - 1][ij]< fMin)
    {
        fMin= fM[nCl - ij - 1][ij];
        iSt_Min= nCl - ij - 1;
        iCl_Min= ij;
    } //if
ShowMessage("Минимальный элемент побочной диагонали \
находится в строке " + IntToStr(iSt_Min + 1) +
            " и столбце " + IntToStr(iCl_Min + 1) +
            ", равен " + FloatToStrF(fMin, ffFixed, 10, 1));
```

Изучите ещё фрагмент программы, который ищет сумму элементов матрицы *над* побочной диагональю.

```
float fS= 0;
for (int ii= 0; ii< nSt; ++ii)
    for (int ij= 0; ij< nCl; ++ij)
        if (ii< nCl - ij - 1)
            fS += fM[ii][ij];
```



```
ShowMessage("Сумма элементов матрицы fM над побочной \
диагональю равна " + FloatToStrF(fS, ffFixed, 10, 2));
```

Рассмотренные выше алгоритмы обработки матриц помогут решению задач, приведенных в следующем разделе. Проявите изобретательность и находчивость, ваши варианты решений будут оригинальнее тех, что даны в разделе 7.7. Проверьте последнее утверждение – доставьте себе такое удовольствие.

Вопросы для самоконтроля

- ☐ Сформулируйте условия нахождения элементов матрицы на главной диагонали.
- ☐ Определите условия нахождения элементов матрицы на отрезке прямой, расположенной над (под) главной (побочной) диагональю.

7.6. Задачи для программирования

Задача 7.1. В матрице $M(6,6)$:

- ☐ в строках с отрицательным элементом на главной диагонали найти наибольший из всех элементов строки и сумму всех таких максимальных элементов;
- ☐ все элементы этой матрицы, которые больше $p=8$ записать в массив Mr ; если в строке нет элемента больше p , то в массив Mr записать 0;
- ☐ определить в каждом столбце количество элементов, которые без остатка делятся на $t=5$.

Задача 7.2. В квадратной матрице из нечетного числа n строк и n столбцов по спирали запишите возрастающий ряд чисел от 1 до $n*n$. Пусть этот ряд начинается с верхнего левого угла матрицы и заканчивается в ее центре. Вид решения для $n=7$ показан на рис. 7.5.

7.7. Варианты решений задач

Проект Задача 7.1.

В файле реализации введены глобальные константа и переменная:

```
//Максимальное число строк и столбцов квадратной матрицы
const int n= 20;
int ix;//Текущее число столбцов таблицы строк
```

Матрица-спираль :

1	2	3	4	5	6	7
24	25	26	27	28	29	8
23	40	41	42	43	30	9
22	39	48	49	44	31	10
21	38	47	46	45	32	11
20	37	36	35	34	33	12
19	18	17	16	15	14	13

Рис. 7.5. Вид решения для $n=7$

Тело функции *FormCreate*:

```
// Называем строки таблицы строк Tab
for (int ii= 1; ii<= n; ++ii)
    Tab->Cells[0][ii]=" стр_" + IntToStr(ii);
// Называем столбцы Tab
for (int ij= 1; ij<= n; ++ij)
    Tab->Cells[ij][0]=" ст_" + IntToStr(ij);
//Устанавливаем текущие значения числа строк и столбцов,
//которое вводится в поле ввода с именем m
Tab->RowCount= StrToInt(m->Text) + 1;
Tab->ColCount= StrToInt(m->Text) + 2;

Tab->Cells[0][0]= "№"; //Называем нулевой столбец Tab
//Запоминаем в ix требуемое число столбцов
ix= StrToInt(m->Text) + 2;
Tab->Cells[ix - 1][0]= "fMa"; //Называем столбец
//Выводим только заголовок таблицы строк Mr
Mr->RowCount= 1;
Mr->Cells[0][0]= "№";
Mr->Cells[1][0]= "Mr";
```

Тело функции *mChange*:

```
//Устанавливаем число столбцов и строк таблицы Tab
Tab->Cells[ix - 1][0]= ""; //Стираем прежний заголовок
ix= StrToInt(m->Text) + 2; //Запоминаем текущее значение в m
Tab->ColCount= ix; //Устанавливаем новое текущее значение
// Называем строки Tab
for (int ii= 1; ii<= n; ++ii)
    Tab->Cells[0][ii]= "стр_" + IntToStr(ii);
//Называем столбцы Tab
for (int ij= 1; ij<= n; ++ij)
    Tab->Cells[ij][0]= " ст_" + IntToStr(ij);
Tab->RowCount= StrToInt(m->Text) + 1;
Tab->Cells[ix - 1][0]= "fMa";
Tab->Cells[0][0]= "№";
//Очистка Tab
for (int ii= 1; ii< ix ; ++ii)
    for (int ij= 1; ij< ix ; ++ij)
        Tab->Cells[ij][ii]= "";
//Выводим только заголовок таблицы Mr
Mr->RowCount= 1;
```

Тело функции *PyskClick*:

```
//Вводим числа m, p и t
const int im= StrToInt(m->Text),
        ip= StrToInt(p->Text),
        it= StrToInt(t->Text);
//Определяем массивы
float fM[n][n], //Исходная квадратная матрица
      fMa[n], //Максимальные элементы строк
      fMa_t[n], //Число элементов столбцов, кратных it
      fMp[n*n]; //Элементы fM, превышающие ip
//Устанавливаем число строк и столбцов таблицы Tab
Tab->RowCount= im + 1;
Tab->ColCount= im + 2;
```

```
//Заносим исходные данные в массив fM[m][m]
for (int ii= 1; ii<= im; ++ii)
    for (int ij= 1; ij<= im; ++ij)
        fM[ii - 1][ij - 1]= StrToFloat(Tab->Cells[ij][ii]);
//Определение наибольшего элемента каждой строки с
//отрицательным элементом на главной диагонали
//Счётчик числа строк с отрицательным
//элементом на главной диагонали
int ir= 0;
float fS= 0;//Для суммирования
for (int ii= 0; ii< im; ++ii)
{
    float fMax;//Максимальный элемент строки
    //с отрицательным элементом на главной диагонали
    if (fM[ii][ii]< 0)
    {
        fMax= fM[ii][0];
        for (int ij= 1; ij< im; ++ij)
            if (fM[ii][ij]> fMax)
                fMax= fM[ii][ij];
        fMa[ir]= fMax;
        Tab->Cells[im + 1][ir + 1]=
            FloatToStrF(fMa[ir], ffFixed, 10, 2);
        fS += fMa[ir];
        ir++;
    }//if
}//for_ii
Mp->RowCount= 1;//Вывод только заголовка таблицы Mp
if (ir> 0)
    ShowMessage("Сумма максимальных элементов строк\
с отрицательными элементами на главной диагонали равна " +
FloatToStrF(fS, ffFixed, 10, 2));
else
    ShowMessage("Все элементы главной диагонали\
больше нуля! ");
//Построение массива fMp
ir= 0;//Счётчик числа элементов массива fMp
for (int ii= 0; ii< im; ++ii)
{
    int ik= 0;//Счётчик числа искомых элементов в строке
    for (int ij= 0; ij< im; ++ij)
        if (fM[ii][ij]>ip)
        {
            fMp[ir]= fM[ii][ij];
            ir++;
            ik++;
        } //if
    if (ik== 0)//Если искомых элементов в строке нет
    {
        ir++;
        fMp[ir]= 0;
    }//if
}//for_ii
//Установка числа строк и столбцов таблицы Mp
Mp->RowCount= ir + 1;
```

```

Mp->ColCount= 2;
Mp->Cells[0][0]= "№";
Mp->Cells[1][0]= "Мр";
//Очистка строки (im + 1) таблицы fM
for (int ij= 1; ij<= im; ++ij)
    Mp->Cells[ij][im + 1]= "";
//Вывод результатов в таблицу Мр
for (int ii= 0; ii<= ir; ++ii)
{
    Mp->Cells[0][ii + 1]= ii + 1;
    Mp->Cells[1][ii + 1]=
        FloatToStrF(fMp[ii], ffFixed, 10, 1);
    fMp[ii]= 0; //Очистка
} //for_ii
//Определяем и выводим в строке 'Mt' таблицы Tab число элементов столбцов,
кратных it
Tab->RowCount= im + 2; //Установка дополнительной строки
Tab->Cells[0][im + 1]= "Mt"; //Вывод её заголовка
for (int ij= 0; ij< im; ++ij)
{
    int iS= 0; //Счётчик
    for (int ii= 0; ii< im; ++ii)
        if (fmod(fM[ii][ij], it)== 0 && fM[ii][ij]!= 0)
            iS++;
    Tab->Cells[ij + 1][im + 1]= iS;
} // for_ij

```

Проект *Задача 7.2.*

Тело функции *FormCreate*:

```

Tab->FixedCols= 0;
Tab->FixedRows= 0;

```

Тело функции *mChange*:

```

//Установка текущего числа столбцов и строк таблицы Tab
Tab->ColCount= StrToInt(m->Text); //m - объект типа TEdit
Tab->RowCount= StrToInt(m->Text);

```

Тело функции *PyskClick*:

```

const int n= 20; //Максимальное число строк и столбцов
//Текущее число строк и столбцов
    im= StrToInt(m->Text),
    ip= (im - 1)/2; //Число витков спирали
float iM[n][n]; //Матрица из n строк и n столбцов
//Установка числа строк и столбцов таблицы строк Tab
Tab->RowCount= im;
Tab->ColCount= im;
int iCh= 0; // текущее число, записываемое в матрицу
//Цикл по виткам спирали
for (int it= 1, ii, ij; it<= ip; ++it)
{ //ii- номер строки, ij- номер столбца
    ii= it - 1; //Верхняя часть витка
    for (ij= it - 1; ij<= (im - it); ++ij)
    {
        iCh++;
    }
}

```

```

        iM[ii][ij]= iCh;
    }//for_ij
    ij= im - it; //Правая часть витка
    for (ii= it; ii<= (im - it); ++ii)
    {
        iCh++;
        iM[ii][ij]= iCh;
    }//for_ii
    ii= im - it; //Нижняя часть витка
    for (ij= im - it - 1; ij>= (it - 1); -ij)
    {
        iCh++;
        iM[ii][ij]= iCh;
    }// for_ij
    ij= it - 1; //Левая часть витка
    for (int ii= im - it - 1; ii>= it; -ii)
    {
        iCh++;
        iM[ii][ij]= iCh;
    }//for_ii
} //for_it
iCh++;
iM[ip][ip]= iCh;
//Вывод матрицы-спирали
for (int ii= 0; ii< im; ++ii)
    for (int ij= 0, ix; ij< im; ++ij)
    {
        ix= iM[ii][ij];
        Tab->Cells[ij][ii]= ix;
    }//for_ij

```



Урок 8. Функции

8.1. Важный инструмент структурирования программ

Для разработки легко читаемых программ рекомендуется строго придерживаться правил хорошего стиля программирования. Напомним уже рассмотренные правила.

- ☐ Названия проектов, идентификаторы констант и переменных должны быть краткими и информативными.
- ☐ В код программы включать лаконичные комментарии о назначении констант, переменных и выполняемых блоков.
- ☐ Разбивать программу на законченные логические блоки при помощи отступов и пропусков строк.
- ☐ Все фрагменты приложения разрабатывать в виде, удобном для последующей модернизации.
- ☐ Создавать программную документацию с включением в неё схем алгоритмов.

Познакомимся с ещё одним очень важным пунктом хорошего стиля программирования заключающимся в том, чтобы приложения разрабатывать с использованием пользовательских функций. В *Builder* и в других аналогичных средах разработки программ (например, *Delphi*) без пользовательских функций обойтись нельзя. В этих средах заготовки функций, предназначенных для обработки событий визуальных объектов, генерируются полуавтоматически. Разработчик программы лишь вписывает в тело созданной заготовки (например, *FormCreate*, *PyskClick*, и др.) необходимый код.

Пользовательские функции применяются не только для обработки событий – любые части программ, в которых осуществляется совокупность каких-либо законченных действий общей задачи, рекомендуется оформлять в виде функции. Функция – это фрагмент общей программы, который можно вызывать произвольное число раз в теле основной программы с разными параметрами (или аргументами) или без них. В качестве аргументов используются переменные, константы и выражения. Иными словами функция – это набор каких-либо действий, обозначенных единым именем. Если разработанная функция оказалось достаточно громоздкой, например, содержит более 50 строк, то желательно её представить в виде совокупности *менее объёмных* функций, каждая из которых решает более простую задачу.

Зачем это нужно? Господа, – это выстраданный предшествующими поколениями программистов путь наиболее *быстрой* разработки приложений. При этом число строк программы часто *существенно* уменьшается (что далеко не самое главное), программу очень удобно отлаживать и модернизировать. Функции из готового, полностью отлаженного приложения, можно использовать при разработке других приложений, где выполняются такие же задачи. Поэтому предлагаемый подход экономит время и силы при разработке программ.

Одним из наиболее *важных* преимуществ пользовательских функций является то, что с их использованием приложение становится хорошо *структурированным*. В структурированной программе *с высоты птичьего полёта* можно рассмотреть код всей программы, легко увидеть пути его улучшения, возможные ошибки.

Действия, предусмотренные вызовом каждой *стандартной* функции, осуществляются при помощи запуска на выполнение её тела – фрагмента программы. Такие блоки программ конструируются по строго определённым правилам. Ниже *детально* рассмотрим правила разработки и применения функций. Однако вначале напомним всё, что уже известно о функциях из предшествующих уроков.

8.2. Что уже известно о функциях

В справочной системе или учебниках (см. также раздел 3.2) *стандартная* функция (функция, описанная в одном из модулей (библиотек) *Builder*), например, косинуса описываются так: *double cos(double dx)*.

Для *каждой* функции приводится тип аргумента и тип возвращаемого функцией значения. При вызове функции её аргументу (или аргументам) могут назначаться *только* допустимые значения. В приведенном примере аргументом функции могут быть значения типа *double* и все вложенные в него типы: *float, int, long int*.

Укажем два важных термина, относящихся к применению функций. В нашем примере аргумент *dx* при *описании* функции называется *формальным* параметром, он может приобретать конкретное или *фактическое* значение *только* при *вызове* функции в программе. Рассмотрим следующий пример:

```
float fx= 0.12, fy;  
fy = cos(fx)*sin(fx);
```

В этом фрагменте аргумент *fx* *применённых* функций является уже *фактическим* параметром. Как уже отмечалось, тип фактического параметра согласовывается с типом формального параметра: должен соответствовать типу формального параметра, указанному при описании *каждой* конкретной функции, или быть вложенным в него.

Ранее многократно использовались функции, которые ничего не возвращают в место их вызова (возвращают тип *void*). Однако они производят действия *процедурного* характера, например, выполняют обработку нажатия кнопок, вывод на экран сообщений и др. Ясно, что упомянутый спектр решаемых задач исключает применение имени таких функций в математических выражениях.

Функции же, возвращающие *числовые* значения, могут выступать в роли опе-

рандов математических выражений, являться элементами формул, например:

```
float fx= 5.12, fy;
fy = 12.5*Primer_Fynkcii(fx);
```

Возвращаемое функцией значение замещает в формуле имя функции. Однако, то значение, которое функция возвращает, можно не использовать:

```
Primer_Fynkcii(fx);
```

Последний вызов функции применяется при осуществлении какого-либо процедурного действия (например, функция *ShowMessage()*), либо когда её аргументом является параметр-переменная (см. раздел 8.4).

Правила применения стандартных и пользовательских функций полностью совпадают. Рассмотрим далее все этапы разработки функций.

8.3. Описание и объявление функций

8.3.1. Тип возвращаемых значений

Синтаксисом допускается *не указывать* тип возвращаемого функцией значения. В этом случае функция возвращает тип *int*. Не рекомендуем использовать такой способ записи, поскольку он является потенциальным источником ошибок.

Тип возвращаемого функцией значения может быть произвольным, *за исключением массива и функции*. Однако типом функции *может быть указатель* на массив или функцию. Указатели и динамические переменные *подробно* рассмотрены на десятом уроке. Сейчас же приводятся только сведения, необходимые для наиболее полного изложения возможностей функций.

Указатель это поименованная ячейка памяти, в которую можно записать *только* адрес ячейки-байта, начиная с которого располагается объект *заданного* типа. Объём любого указателя составляет 4 байта. Объектом может быть динамический объект или динамическая переменная, простая или структурированная переменная, пользовательская функция. Объектом переменной (обычной или динамической) является *группа* ячеек памяти, в которых эта переменная размещается. Визуальные и неvizуальные объекты также расположены в блоках памяти. Адреса начальных ячеек этих блоков хранятся в указателях на эти объекты. Объявляется указатель *Yk* на тип, например, *int* вот так:

```
int *Yk; //Объявлен указатель Yk на тип int
```

Тип *int* указателя *Yk* позволяет в ячейку памяти (с именем *Yk*) записать адрес *только* переменной с упомянутым типом. Запись такого адреса осуществляется с использованием операции *взятия адреса* &. Символ амперсанта «&», поставленный перед переменной, определяет её адрес (адрес начального байта объекта переменной), который затем записывается в переменную-указатель *Yk*. При помощи этой операции определяется адрес статической переменной любого типа.

```
int iy= 58; //Объявлена и инициализирована переменная iy
Yk= &iy; //Адрес переменной iy записан в указатель Yk
```


Значение, записанное в *iy* можно теперь изменить и просмотреть так:

```
*Yk= 59;  
ShowMessage (*Yk) ; //59
```

Форма записи **Yk* означает обращение к *объекту* указателя, к ячейкам памяти, в которых расположено целочисленное значение 59, адрес начального байта этой группы ячеек записан в указателе *Yk*.

Программист в выбранном им месте программы может при помощи операции *new* породить в памяти *динамическую* переменную заданного типа (выделить блок ячеек в динамической памяти) и адрес его начальной ячейки записать в указатель *Yk*:

```
int *Yk;  
Yk= new int; //Адрес динамической переменной записан в Yk
```

Рекомендуется совмещать операции объявления указателя и его инициализацию (запись в него конкретного адреса). В этом случае объявление указателя на объект типа *int* и выделение места в динамической памяти для его размещения осуществляется следующим образом:

```
int *Yk = new int;
```

После такой операции в ячейке с именем *Yk* будет храниться адрес ячейки, расположенной в динамической памяти компьютера, начиная с которой *может* располагаться сам объект динамической переменной заданного типа (объект *можно* записать в эти ячейки). Под размещение объекта динамической переменной выделяется память, равная размеру её типа. Когда в ходе дальнейшей работы программы объект *Yk* (он называется динамическим объектом) окажется ненужным, его рекомендуется уничтожить – освободить память, занятую для его размещения:

```
delete Yk; //Уничтожение указателя Yk
```

Этой операцией *разрывается* связь ячейки *Yk* с адресом начального байта группы ячеек, в которых помещён объект (тело) динамической переменной. При завершении работы приложения все его динамические объекты настоятельно рекомендуется *уничтожать*. В этом случае на освобождённых ресурсах оперативной памяти появляется возможность размещения *новых* объектов переменных произвольных типов и объектов. Это скорее не признак хорошего стиля программирования, а важное правило программиста, использующего динамические переменные и объекты.

Операции *new* (породить) и *delete* (уничтожить) – это стандартные операции заголовочного файла *stdlib.h*, который как отмечалось на третьем уроке, подключается к любому проекту *автоматически*.

8.3.2. Если функция не имеет параметров

Функция может не иметь параметров. В этом случае в заголовке функции после её имени в круглых скобках следует указать служебное слово *void*:

```
void Primer_Fyn(void)
```

Скобки можно оставить также пустыми:

```
void Primer_Fyn()
```

Однако это менее удачный вариант. Дело в том, что в языке *C* пустые скобки означают аргумент произвольного типа. Поэтому если потребуется перенести проект из *Builder* на этот архаичный язык, то могут возникнуть трудности. Рекомендуем *всегда* использовать *только* первый, более строгий, вариант.

Функции без параметров, использующие *глобальные* переменные, могут порождать трудно определяемые ошибки. Поэтому применение таких функций является очень *плохим* стилем программирования.

8.3.3. Прототип функции

Прототип функции представляет собой заголовок функции с точкой и запятой «;» в конце. Если в файле реализации описана пользовательская функция (правила описания излагаются позже), то её можно вызывать *произвольное* число раз во всех других пользовательских функциях, описанных *ниже* упомянутой функции. В этом случае прототип функции можно не использовать, *Builder* и без него узнает вызванную функцию, проконтролирует правильность следования типов её фактических параметров и их количество.

Если же пользовательская функция применяется *до* её *описания* или же её код находится в ином файле (модуле), в таких случаях перед вызовом функции в файле реализации включается прототип функции – её предварительное объявление. В противном случае в ходе компиляции программы будет выведено сообщение об ошибке, связанной с вызовом *неизвестной* функции.

В прототипе можно не указывать *имена* параметров, поскольку при компиляции они не используются, важен лишь тип этих параметров, последовательность типов и количество параметров. Однако при реализации такого варианта описания прототипа текст программ становится менее ясным. Ниже приводятся примеры обоих вариантов описания прототипов функций:

```
float Primer_Fynkcii (float fx);  
float Primer_Fynkcii (float); // Менее предпочтительный  
                             // вариант описания прототипа
```

Рекомендуется прототипы функций и их комментарии размещать в *начале* модуля. Это делает программу более наглядной и самодокументированной.

8.3.4. Переменное число параметров функции

Можно разрабатывать функции, принимающие *переменное* число параметров или параметры, типы которых заранее *неизвестны*. В функциях вместо неизвестного числа параметров или параметров неизвестного типа ставится многоточие «...»:

```
int Prim_Fyn (double dx, ...); // Определён тип только первого  
                             // параметра
```

Запятая перед многоточием необязательна.

```
int Prim_F (...); //Принимает произвольное число параметров
                  //произвольного типа
```

Не указывать параметр (или параметры) можно *только* в конце списка. Пропускаемый параметр (или параметры) не может располагаться в середине или в начале списка параметров, поскольку, в ходе компиляции все параметры, находящиеся справа от пропущенного параметра, также *пропускаются*. Это нарушает логику работы приложения, поэтому при компиляции выводятся сообщения о совершенно неадекватных ошибках.

Приведём пример функции с *переменным* числом параметров. Функция *Proizv* находит произведение своих фактических параметров, число которых может быть произвольным. Фактические параметры вводятся в стек в порядке их перечисления в вызванной функции. Значения даже целочисленных параметров *следует* вводить с использованием математической точки (см. код приложения), поскольку в этом случае *все* фактические параметры будут иметь *один и тот же* тип *double* (8 байт). Согласно применённому алгоритму одинаковый объём ячеек для всех параметров позволяет объявленный указатель с именем *Ykazat* передвигать с одного параметра на другой (подробнее см. десятый урок). При *каждом* перемещении указатель сдвигается на 8 байт (на размер объекта для использованного типа аргументов). В приложении, которое приведено ниже, указатель *Ykazat* инициализируется с использованием операции взятия адреса.

```
//Файл реализации
double Proizv (double dx, ...) //Заголовок функции
{
    double Proizvedenie= 1.0;
    //Настраиваем указатель на первый параметр: адрес первого
    //формального параметра присваиваем переменной-указателю
    double *Ykazat= &dx;
    //Если в объекте переменной содержится 0.0, то
    //функция возвращает нулевое значение
    if (*Ykazat== 0.0)
        return 0.0;
    //Цикл умножения продолжается до тех пор, пока в объекте
    //указателя *Ykazat не окажется нулевое значение 0.0
    for ( ; *Ykazat; Ykazat++)
        Proizvedenie *= *Ykazat;
    return Proizvedenie;
} //Proizv

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    double dx= Proizv(0.0);
    ShowMessage(dx); //0
    dx= Proizv(2.0, 3.0, 0.0);
    ShowMessage(dx); //6
    dx= Proizv(5.0, 3.0, 2.0, 0.0);
    ShowMessage(dx); //30
} //Button1Click
```

8.3.5. Параметры со значениями по умолчанию

Параметры со значениями по умолчанию должны указываться при *первом* упоминании имени функции (обычно в прототипе). Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций. Приведём пример функции с параметрами по умолчанию:

```
double Fun(double dx1= 1, double dx2= 2, double dx3= 3)
{
    return dx1 + dx2 + dx3;
} // Fun
```

Если при вызове функции параметр по умолчанию не указан, то в функцию передаётся его значение по умолчанию:

```
double dF= Fun();
```

В нашем примере в переменной *dF* будет находиться значение 6 ($1 + 2 + 3 == 6$).

В случаях

```
double dF1= Fun(2), // (2 + 2 + 3 == 7)
       dF2= Fun(2, 3); // (2 + 3 + 3 == 8)
```

переменные *dF1* и *dF2* содержат соответственно числа 7 и 8.

При оформлении заголовков функции параметры по умолчанию должны быть самыми правыми (последними) в списке аргументов функции. При вызове такой функции пропускать параметр (или параметры), у которого нет значения по умолчанию, нельзя:

```
double Fun1(double dx1, double dx2= 2, double dx3= 3)
{
    return dx1 + dx2 + dx3;
} // Fun1
double dF1= Fun1(2), //В dF1 содержится 7 == 2 + 2 + 3
dF2=Fun1(2, 3); //В dF2 содержится 8 == 2 + 3 + 3
```

8.3.6. Чем функции отличаются от программ

Функции и программы по своей структуре очень похожи. Судите сами, программы и функции могут содержать свои разделы с описанием констант, пользовательских типов и переменных. Однако если в файле реализации программ могут объявляться и описываться пользовательские функции, то в теле функций *не допускается* объявление прототипов и описание других пользовательских функций. Запомните господи, что функции *не могут* быть *вложенными* друг в друга. В пользовательской функции разрешается вызывать те функции, которые описаны выше в файле реализации или их прототип указан перед разрабатываемой функцией.

8.3.7. Операция расширения области видимости

Если имя локальной переменной (объявленной в теле функции) *совпадает* с идентификатором глобальной переменной (объявленной в файле реализации, вне тела рассматриваемой функции), то глобальная переменная становится *невидимой* или *недоступной* в функции. Получить доступ к такой глобальной переменной можно только при помощи операции «::*» расширения области видимости (действия):*

```
ix = ::ix + 5;
```

В этом примере значение глобальной переменной *ix* складывается с числом 5, после чего результат суммирования записывается в локальную переменную *ix*. Операцию «::*» ещё называют уточнением доступа к области видимости (действия), указанием области видимости.*

Рекомендуем воздерживаться от применения одноимённых глобальных и локальных переменных. Читать и отлаживать приложение значительно легче и удобнее, когда локальные и глобальные переменные, близкие по назначению, имеют похожие, но не совпадающие имена.

8.3.8. Выход из функции

Если функция *не возвращает* какое-либо значение (возвращает тип *void*), то выход из неё происходит по достижению закрывающей её тело фигурной скобки или при выполнении оператора *return* (возврат), который прерывает выполнение функции и возвращает то значение, которое после него поставлено. Это значение может быть результатом математического выражения, вызова переменной, функции, именованной или неименованной константы. Первый вариант выхода из функции неоднократно использовался ранее, например, при реализации функции, вызываемой нажатием командной кнопки.

В случае, когда функция *возвращает* значение, нормальный выход из неё (не аварийный) осуществляется оператором *return*:

```
return выражение;  
return вызов функции;  
return константа;  
return переменная; // Более предпочтительный способ
```

При этом типы выражения, функции, константы и переменной должны быть согласованными с типом, который возвращает функция:

```
float Symma (float fx, float fy)  
{  
    float fz= fx + fy;  
    return fz;  
} // Symma
```

Все операторы, поставленные за оператором *return*, выполняться никогда не будут.

Прервать выполнение функции можно как оператором *return*, так и генерацией какого-либо исключения (при помощи блока *try-catch*) либо применением молчаливого исключения (без сообщений) с использованием функции *Abort()*.

8.3.9. Запрет и разрешение доступа к функции

Если функция объявлена со спецификатором *static*, например:

```
static void Fynkc(void);
```

то её можно использовать только в файле, в котором она описана, в иных файлах-модулях её вызов невозможен.

Спецификатор *extern* предполагается по умолчанию, поэтому его использование излишне. К функциям с таким спецификатором можно получить доступ в тех модулях-файлах приложения, в которых указан их прототип.

8.4. Различные способы передачи параметров в функции

8.4.1. Передача параметра по значению

Значения параметров (аргументов) функции могут *не возвращаться* в программу (если они и претерпели изменение в ходе работы этой функции). В этом случае способ передачи параметров в функцию называется *передачей параметров по значению*. Каждый такой параметр *вначале* копируется во временную память, выделяемую только на время работы функции (в стек). В стеке хранятся и значения локальных переменных, и код самой функции. Переменные, переданные функции в качестве параметров-значений, являются её *локальными* переменными.

Примером *стандартной* функции с параметром-значением является *ShowMessage*(“Сообщение”), которая показывает переданный ей параметр *Сообщение* в специальном окне. Приведём ещё два примера стандартных функций с параметрами-значениями. Функция *sqr*(*Chislo*) возвращает корень квадратный из числа *Chislo*, функция *pow*(*Chislo*, *Stepenj*) возвращает результат возведения числа *Chislo* в произвольную степень *Stepenj*. Если аргументами упомянутых функций являются переменные (конечно, глобальные для функции), то после вызова таких функций эти переменные *не изменятся* (они не *могут* изменяться). Поэтому в качестве параметров-значений используются константы, переменные и математические выражения. При этом типы формальных и фактических параметров должны быть согласованными. При разработке *пользовательских* функций применим тот же механизм, который реализован в стандартных функциях с параметрами-значениями. Ниже изложим его сущность.

При описании параметров-значений в заголовках функций и в их прототипах следует *перед* именем параметра указать *только* тип параметра. Если на место формального параметра-значения поставлено математическое выражение, то

вначале оно вычисляется, полученное значение копируется во временную память и только затем оно (из стека) передаётся функции. В случае, когда таким параметром является глобальная переменная или константа, то значение параметра *сразу* копируется из сегмента данных в сегмент стека. Таким образом, в случае применения переменных и констант значение фактического параметра находится одновременно в *двух* местах: в сегменте данных (оригинал) и в сегменте стека (копия). Функция с параметром-значением может изменять *только* значение фактического параметра, находящегося в стеке, после такой обработки это значение из функции *не передаётся*, оно остаётся в стеке и исчезает при вызове другой функции или повторном вызове прежней функции.

В этом способе передачи параметра глобальная переменная *защищена* от неконтролируемых изменений, однако, при этом дополнительно затрачивается время на создание копии, увеличивается объём оперативной памяти, выделяемой на размещение копии параметра в стеке. В некоторых случаях стек может оказаться переполненным. Все упомянутые преимущества и недостатки проявляются *только* когда используются переменные очень *больших* объёмов, например, структуры. Массивы передать в функцию в качестве параметра-значения невозможно, в функцию передаётся *только* адрес массива. Различные способы передачи параметров по адресу рассматриваются в следующих пунктах.

8.4.2. Передача параметра по ссылке

Передать программе результат работы функции можно не при помощи значения, которое она возвращает в место своего вызова, а с использованием её параметра или параметров (аргументов). При этом фактические значения параметров могут иметь исходные значения или быть неопределёнными перед вызовом такой функции. Для реализации упомянутого способа передачи параметра в *Builder* имеется целый набор средств. Все они основаны на том, что в функции *осуществляется прямой доступ* к глобальной статической переменной или объекту динамической переменной. Этот доступ осуществляется передачей в функцию (в стек) *адреса* переменной, (занимающего лишь 4 байта памяти). В результате функция в ходе своей работы изменяет непосредственно *глобальную* переменную или *глобальный* объект.

Поскольку эти параметры *могут* изменяться в ходе работы функции, то их фактические параметры *не могут* быть константами и математическими выражениями, фактическими параметрами в этом случае должны быть *только переменные*. Не случайно параметры, передаваемыми по адресу, ещё называют *параметрами-переменными*.

При использовании громоздких переменных (занимающих большие объёмы оперативной памяти) в качестве параметров-переменных производительность приложения существенно повышается. Это обусловлено тем обстоятельством, что *не требуется* расходовать время на копирование данных из сегмента данных в сегмент стека. Кроме того, данные находятся *только* в одном месте, поэтому уменьшается расход оперативной памяти. Конечно, при таком механизме переда-

чи параметра программист должен взять на себя *всю* ответственность за целостность данных, с которыми работает программа.

Рассматриваемый механизм передачи параметра реализуется двумя способами: с помощью ссылок и при помощи указателей. В соответствии с заголовком данного пункта, рассмотрим вначале вызов параметра по ссылке.

Ссылочный параметр – это псевдоним или синоним основного имени, т.е. *дополнительное имя* фактического параметра. Выбранный параметр функции передаётся по ссылке, если после типа параметра (перед именем параметра) в прототипе и заголовке функции указать символ амперсанта «&». Перед амперсантом и после него могут ставиться пробелы:

```
int &ixx
int & ixx
int& ixx
```

Выше приведены три *эквивалентные* ссылки на тип *int*. Именно так и говорят «ссылка на тип», а не «ссылка на параметр». Этим подчёркивается, что *адрес* любого фактического параметра *указанного типа* будет передан в функцию, а функция заранее побеспокоится о том, чтобы выделить в стеке ячейку в 4 байта для размещения соответствующего адреса. Последующие ячейки стека используются для размещения *других* параметров и локальных переменных.

При вызове функции знак амперсанта у ссылочного параметра не используется. Проиллюстрируем упомянутые синтаксические правила простым примером.

```
void Kvadrat(int &ixx); //Прототип функции вычисления квадрата
                        //её аргумента, имя аргумента можно не указывать
void Kvadrat(int &ix) //Заголовок функции
{
    ix= ix*ix; // или   ix *= ix;
} // Kvadrat
```

Фрагмент кода, который обеспечивает вызов такой функции:

```
int iy= 2;
Kvadrat(iy); //Символ & не используется, iy== 4
```

8.4.3. Передача параметра по адресу

Второй способ передачи в функцию адреса переменной связан с использованием указателей. Чтобы функция работала с *адресом* переданного ей фактического параметра (переменной) в описании и в прототипе такой функции этот параметр объявляют *указателем*. Как отмечалось выше, такая операция выполняется с использованием символа «*». Сама эта операция называется операцией *косвенной адресации*, операцией *разыменования*, операцией доступа по адресу или обращения по адресу. Нам представляются первые два термина крайне неудачными. Однако они часто используются в литературе, поэтому необходимо понимать их смысл. Приведём поясняющий пример.

```
void Kvadrat(int *ixx); //Прототип функции вычисления квадрата
                        //её аргумента
```



```
void Kvadrat(int *ix) //Заголовок функции
{
    *ix = *ix**ix; //или *ix *= *ix;
} // Kvadrat
```

Напоминаем, в теле функции «**ix*» означает объект переменной, иными словами значение (содержимое) переменной, которое находится в блоке ячеек оперативной памяти, на первую из которых направлен указатель *ix*. Если же передаваемый в функцию параметр является визуальным объектом, то в теле функции его имя используется *без звёздочки* слева. Дело в том, что в именах визуальных объектов уже находятся адреса объектов этих объектов (осознаём запутанность последнего утверждения). Работа с визуальными объектами, передаваемыми в функцию по адресу, иллюстрируется отдельным учебным приложением (см. раздел 8.7). Сейчас же покажем вызов функции с параметром-переменной простого типа:

```
int iy= 2;
Kvadrat(&iy); //iy== 4
```

Как видите, при вызове функции с параметром-указателем в качестве её фактического параметра должна передаваться не сама переменная, а её адрес, получаемый с помощью операции взятия адреса &.

Читатель, видимо, до конца не осознал сущность различий между параметром, передаваемым по ссылке и по адресу. Если это так, то это вполне нормальное явление и беспокоиться не стоит: тонкости различия при первом знакомстве с этими механизмами уяснить весьма трудно. После рассмотрения учебной программы (см. раздел 8.7) ситуация прояснится. Сейчас важно запомнить *синтаксис* применения каждого способа передачи параметра, поскольку ниже познакомимся с ещё более запутанным синтаксисом.

8.4.4. Применение спецификатора *const* в ссылочных параметрах

Если в заголовке функции перед параметром-ссылкой поставить ключевое слово *const*, то аргумент в функцию передаётся как константа. Это означает, что в функцию *посылается адрес* переменной, однако функция *не может изменить* значение этой переменной. Такой механизм обеспечивает целостность данных и вместе с тем позволяет избавиться от копирования громоздких переменных, типа больших структур, что направлено на повышение быстродействия программы. Согласно синтаксису в *прототипе* таких функций спецификатор *const* можно не указывать. Однако мы *настоятельно* рекомендуем для большей ясности программы использовать *полный* вариант описания прототипа:

```
double Fynkcija(const &x);
```

Тип константной ссылки в заголовке функции и в прототипе можно не указывать.

8.4.5. Применение спецификатора *const* в параметрах-указателях

Имеется три варианта применения спецификатора *const*: неконстантный указатель на константные данные; константный указатель на неконстантные данные; константный указатель на константные данные. Внесём ясность в эти термины подробным их рассмотрением.

1. Неконстантный указатель на константные данные – это указатель, который можно перестраивать, чтобы указывать на любые переменные выбранного типа (при помощи операции взятия адреса переменной), но сами данные, на которые он указывает, не могут изменяться:

```
void Fun(const int *iYkaz);
```

В этом прототипе объявляется функция *Fun*, в теле которой такой указатель можно перемещать с одной переменной типа *int* на другую (как это можно осуществить подробно рассказывается в десятом уроке), но сами эти переменные изменить нельзя (будет выведено сообщение об ошибке при компиляции программы). Рассмотренный механизм даёт возможность защитить исходные значения параметров от несанкционированного изменения.

2. Константный указатель на неконстантные данные – это указатель, который всегда указывает на одну и ту же ячейку памяти, данные в которой можно изменять. Этот вариант, например, реализуется по умолчанию для имён массива. Имя массива – это константный указатель на начало массива. Используя имя массива и его индексы можно обращаться ко всем данным в массиве и изменять их. Прототип функции с таким механизмом передачи параметра:

```
void Fun(int *const iYkaz);
```

3. Константный указатель на константные данные всегда указывает на один и тот же блок памяти с неизменными значениями его ячеек. Такой механизм следует, например, применить в случае, когда необходимо передать массив функции, которая *только читает* значения его ячеек, используя индексы массива, но сами значения не изменяет. Пример прототипа для этого случая:

```
void Fun(const int *const iYkaz);
```

Этот механизм предоставляет наименьший уровень доступа к данным.

8.5. Перегрузка функций

Функции с одним и тем же именем в одном и том же приложении при определённых условиях *могут* выполнять совершенно *разные* задачи. Применение одноимённых функций становится удобным, например, при решении *похожих* задач с *разными* типами данных. В этом случае программа становится легко читаемой.

Как же осуществляется выбор *нужной* функции в каждом *конкретном* случае? Это оказывается возможным по *типу, количеству, последовательности следования типов фактических* аргументов функции. Такие функции именуются *перегруженными* (от *overloading functions*), а механизм их разработки – *перегрузкой функций*. Упомянутые термины общеприняты, однако очень уж неудачны. Более понятным названием механизма является *переопределение функций*. О справедливости такого заключения судите сами после практического знакомства с перегрузкой-переопределением функций на примере учебной задачи.

Ниже приведен файл реализации приложения *Перегрузка*, в котором перегруженная функция позволяет складывать или соединять значения переменных *разных* типов.

```
int Symma(int ixx, int iyy) {return ixx + iyy;}
float Symma(int ixx, float fyy) {return ixx + fyy;}
float Symma(float fxx, int iyy) {return fxx + iyy;}
float Symma(float fxx, float fyy) {return fxx + fyy;}
String Symma(String sxx, String syy) {return sxx + syy;}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int ix= 1; float fy= 2.2;
    ShowMessage(Symma(ix, ix)); //2= 1+1
    ShowMessage(Symma(fy, ix)); //3.2= 2.2 + 1
    ShowMessage(Symma(ix, fy)); //3.2= 1 + 2.2
    ShowMessage(Symma(fy, fy)); //4.4= 2.2 + 2.2
    ShowMessage(Symma("Привет, ", "Вася!")); // Привет, Вася!
} //Button1Click
```

Перегруженные функции могут иметь параметры по умолчанию. В этом случае значение одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.

Функции не могут быть перегружены, если описание их параметров отличается только модификатором *const* (например, *int* и *const int*) или использованием ссылки (например, *int* и *int&*).

8.6. Шаблоны функций

При решении сходных задач для *разного* типа данных применяются не только перегруженные функции, но и механизм *шаблонов функций*. С применением шаблонов функций код программы становится *значительно* компактнее. Если в механизме перегрузки функций для *каждого* типа данных имеется *свой* код функции, то в механизме шаблонов код функции с формальными параметрами типа данных записывается лишь *один* раз. Шаблон *семейства* функций служит для *автоматической* генерации конкретных определений функций. Это оказывается возможным по *типам* фактических параметров шаблонной функции, которая вызывается в конкретном случае. Аналогично тому, как фактический параметр функции замещает формальный параметр в коде функции, так и в шаблонной функции фактические типы данных, подставленных в шаблонную

функцию при её вызове, заменяют её формальные параметры типа.

Перечислим операции, составляющие сущность механизма шаблонов. По определённым грамматическим правилам для *произвольного* типа данных определяются все действия, выполняемые в функции. Такое определение называется *шаблоном семейства* функций. Шаблон состоит из двух частей: из *заголовка* шаблона с перечислением произвольных типов данных и *шаблонной функции*. В заголовке шаблонной функции типы параметров, упомянутые в шаблоне, *обязательно* должны использоваться. При вызове шаблонной функции (по указанному шаблону) генерируется код функции с типом фактических параметров указанных при вызове шаблонной функции.

Весь текст от начала этого раздела до настоящего места после рассмотрения примеров *рекомендуем* прочитать *повторно* (или более).

8.6.1. Параметры одинакового типа

Рассмотрим вначале шаблон (*template*) функции, возвращающей сумму двух передаваемых в неё параметров *одного произвольного* типа *Tip*:

```
//Шаблон функции Symma_1 в файле реализации
template <class Tip>
Tip Symma_1 (Tip x, Tip y){return x + y;}
```

В функции *Button1Click* по этому шаблону генерируются *различные* функции, результат вычислений приведен в комментариях.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int i1= 1, i2= 2;
    float f1= 1.5, f2= 2.5;
    double d1= 1.55, d2= 2.55;
    String s1= "Привет, ", s2= "Вася!";
    ShowMessage(Symma_1(i1, i2)); //1 + 2= 3
    ShowMessage(Symma_1(f1, f2)); //1.5 + 2.5= 4
    ShowMessage(Symma_1(d1, d2)); //1.55 + 2.55= 4.1
    ShowMessage(Symma_1(s1, s2)); // Привет, Вася!
} // Button1Click
```

Таким образом, при первом вызове функции *Symma_1* согласно типу фактических параметров в шаблонной функции тип *Tip* был замещён на тип *int*, при втором вызове – на тип *float*, при третьем – на тип *double*, при четвёртом – на тип *String*.

8.6.2. Параметры разного типа

В списке параметров шаблона функций может быть *несколько* параметров *разного* типа:

```
template <class Tip_1, class Tip_2, class Tip_3>
Tip_3 Symma_2 (Tip_1 x, Tip_2 y, Tip_3 z)
    {return x + y + z;}
```

Вызовем эту шаблонную функцию с ранее инициализированными переменными:

```
ShowMessage(Symma_2(i1, f1, d1)); // 1 + 1.5 + 1.55 = 4.05
```

8.6.3. Параметры разного формального типа и конкретного типа

В заголовке шаблонной функции, помимо параметров произвольных типов, указанных в шаблоне, могут быть приведены и конкретные типы переменных (или объектов):

```
template <class Tip_1, class Tip_2, class Tip_3>
Tip_3 Symma_3 (Tip_1 x, Tip_2 y, Tip_3 z, int ix)
{ return x + y + z + ix; }
```

В этом случае тип последнего параметра в вызванной функции должен быть согласован с типом завершающего параметра в заголовке шаблонной функции:

```
ShowMessage(Symma_3(i1, f1, d1, i2)); // 1 + 1.5 + 1.55 + 2 = 6.05
```

Тип возвращаемого функцией значения может не параметризоваться, а быть конкретным стандартным или пользовательским типом.

8.6.4. Необходимость применения всех типов объявленных параметров

Как отмечалось выше, *все типы* шаблона функций следует *использовать* в заголовке шаблонной функции. Для пояснения этого ограничения приведём ошибочное определение шаблонной функции:

```
template <class Tip_1, class Tip_2, class Tip_3>
Tip_3 Symma_4 (Tip_1 x, Tip_2 y)
{ return Tip_3 z = x + y; }
```

При компиляции шаблонной функции

```
ShowMessage(FloatToStr(Symma_4(i1, f1)));
```

будет выведено сообщение об ошибке: нельзя найти соответствие между типами переменных шаблона и вызванной шаблонной функции. В заголовке шаблонной функции среди параметров функции *нет* параметра с типом *Tip_3*. Применение этого типа в качестве возвращаемого значения и для определения переменной *z* совершенно недостаточно.

Если же в заголовке шаблонной функции указать параметры *всех* типов шаблона, но некоторые из них не использовать в теле функции, то это не является ошибкой, в ходе компиляции будет лишь предупреждение, что объявленная переменная нигде не используется. Такое *резервирование* параметров в заголовке функции полезно при последующей модернизации приложения. Ниже приведём шаблон и вызов шаблонной функции для последнего случая.

```
template <class Tip_1, class Tip_2, class Tip_3>
Tip_2 Symma_5 (Tip_1 x, Tip_2 y, Tip_3 z)
{ return x + y; }
ShowMessage(FloatToStr(Symma_5(i1, f1, 5.5))); // 1 + 1.5 = 2.5
```

8.6.5. Другие возможности и ограничения шаблонов

1. Имена параметров шаблона *не могут* повторяться в конкретном шаблоне.
2. Имена параметров шаблона *видимы* в теле функции. Поэтому, если идентификаторы аргументов в шаблоне совпадают с именами глобальных объектов, то такие объекты становятся *невидимыми* в шаблонных функциях. Для доступа к ним следует использовать операцию расширения области видимости, но лучше не порождать такую ситуацию.
3. При вызове шаблонных функций могут использоваться как стандартные, так и пользовательские типы данных. При использовании шаблонов функций возможна перегрузка как шаблонов, так и функций. С помощью шаблона *может создаваться функция* с таким же именем, что и *явно определяемая функция*. Распознавание конкретного вызова осуществляется по типам, порядку и количеству параметров функции.
4. Прототип шаблона функций используется в тех же случаях, что и для обычных функций.
5. В отличие от *C++Builder 5.0* в *C++Builder 6.0* поддерживается ключевое слово *typename* вместо слова *class*: при определении шаблонов слово *class* можно заменить словом *typename*:

```
template <typename Tip>
Tip Symma (Tip x, Tip y) { return x + y; }
```

С использованием нового ключевого слова код программы становится более ясным. Дело в том, что объявленный тип параметра совершенно не обязательно должен быть классом, как можно заключить при употреблении ключевого слова *class*.

6. Чрезмерное использование шаблонов может привести к снижению производительности *при разработке* приложений, поскольку замедляется процесс компиляции. Кроме того код отдельной версии функции или класса, сгенерированных по шаблону, дублируется в объектные файлы *всех* модулей, где используется данный класс или функция. Это приводит к разбуханию объектного кода программы, что может стать причиной замедления процесса компоновки. Следует отметить, что с использованием современных быстродействующих компьютеров все эти недостатки не так уж и велики. Однако сообщим итоговый вывод *профессионалов* [12]: шаблоны хороши с *теоретической точки зрения*, но их непросто использовать на практике.

8.6.6. Пример практического применения шаблона

Требуется определить количество нулевых элементов одномерного массива произвольного числового типа. Используем шаблон, определяющий семейство функций, каждая из которых решает поставленную задачу для фактического типа данных. В приложении шаблонная функция возвращает не шаблонный, а заранее определённый тип данных *int*.

Необходимые для проверки программы массивы объявляются и инициализируются в теле функции *Button1Click*. В приложении используется операция *sizeof* – операция определения объёма переменной или типа. Эта операция может применяться с использованием и без использования круглых скобок:

```
int ix;  
1 ix= sizeof(ix); //ix== 4  
2 ix= sizeof int; //ix== 4
```

Для имени массива (например, *iMassiv* или *fMassiv*) операция *sizeof* определяет объём памяти, выделенный для размещения *всего* массива. Применение этой операции для выбранной ячейки массива, например, с номером 0 (*sizeof(iMassiv[0])*), позволяет установить её объём и, следовательно, объём типа массива. Отношение объёмов всего массива и отдельной его ячейки определяет число элементов массива *in*. Например: *int in= sizeof(iMassiv)/sizeof(iMassiv[0])*);

Для явного преобразования типа, например, из типа *float* в тип *int* можно использовать два *совершенно равноправных* варианта:

```
float fy= 5.555;  
1 int ix= (int) fy;  
2 int ix= int (fy);
```

В приложении полагается, что нулевое значение вещественного числа определяется с точностью до третьего знака после математической точки. Поэтому, если после преобразования

```
int ix= (int) fy*1000;
```

оказывается, что *ix==0*, то полагается, что в *fy* записано нулевое значение.

Более подробные комментарии приведены после кода приложения.

В файле реализации описывается шаблон функции для подсчёта нулевых элементов в массиве:

```
template <class Tip>  
int Podschet_0(int iRazmer, Tip *Mas)  
{  
    int ik= 0; //счётчик числа нулевых элементов массива  
    for (int ij= 0; ij< iRazmer; ij++)  
        //if (int (Mas[ij]*1000)== 0) //Вариант преобразования типа  
        if ((int) (Mas[ij]*1000)== 0) //Преобразование типа  
            ik++;  
    return ik;  
} //Podschet_0  
  
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    //Объявление целочисленного массива и его инициализация  
    int iMassiv[]= {1, 0, 6, 0, 4, 10};
```

```
// Определяем число элементов массива iMassiv
int in= sizeof(iMassiv)/sizeof(iMassiv[0]);
ShowMessage("Число нулевых значений равно " +
    IntToStr(Podshet_0(in, iMassiv)));// in==2

//Задание вещественного массива и его инициализация
float fMassiv[]= {1.2, 0.0, 0.2, 0.0, 4.2, 10.2, 0, 1.3};
// Определяем число элементов массива fMassiv
in= sizeof(fMassiv)/sizeof(fMassiv[0]);
ShowMessage("Число нулевых значений равно " +
    IntToStr(Podshet_0(in, fMassiv)));//in==3
} //Button1Click
```

Для целочисленного массива число нулевых значений равно 2, а для вещественного – 3. В шаблонную функцию *Podshet_0* передаётся два параметра: первый из них – это *нешаблонный* параметр типа *int* (его значение определяется в теле функции *Button1Click*), в качестве второго параметра передаётся переменная *шаблонного* типа, она объявлена указателем на шаблонный тип, в нашем примере её фактическими параметрами являются идентификаторы массивов разных типов.

Здесь приведен один из способов передачи массивов в функцию в качестве её фактического параметра. Всего имеется восемь способов этого процесса, четыре из них будут подробно рассмотрены в следующем разделе. Фактическим параметром функции *Podshet_0* выступает массив, имя же *любого* массива является указателем, поэтому при передаче массива в функцию в качестве её фактического параметра адрес массива при помощи операции взятия адреса «&» не определяется.

8.7. Иллюстрация применения пользовательских функций

Проиллюстрируем применение пользовательских функций на задачах, рассмотренных ранее. Поскольку алгоритмы их решения уже известны, поэтому основное внимание уделим *синтаксису* разработки пользовательских функций. Перечень задач, решаемых новым учебным приложением, приведен ниже.

1. Ввести на экран исходные массивы.
2. Инициализировать целочисленные одномерные массивы *n_Mas* и *m_Mas*, состоящие соответственно из *n* и *m* элементов.
3. Отсортировать массивы *n_Mas* и *m_Mas* по возрастанию значений их элементов методом линейной сортировки.
4. Вывести на экран массивы после их сортировки.
5. Вычислить и вывести на экран сумму сумм компонентов исходных массивов.

Интерфейс приложения *Пользовательские функции* показан на рис. 8.1. Для ввода исходных массивов и вывода результатов их сортировки на интерфейсе имеются две *Таблицы Строк* соответственно с именами *Isx_Mas* и *Sort_Mas*. Полям ввода длин массива присвоены имена *Dl_n_Mas* и *Dl_m_Mas*. Поле вывода



Рис. 8.1. Интерфейс приложения *Пользовательская функция*

суммы сумм массивов имеет имя *Symma_Symm*. Код приложения и его подробный комментарий помещаются ниже.

Фрагмент файла реализации.

```
const int nn= 100; //Максимальная длина массива
//Определение пользовательского типа
typedef int int_Mas[nn];

//Заносит исходные данные из таблицы в массив
void Vvod_Mas(TStringGrid *Is_Mas, int_Mas & iMas,
              int iDl, int iStolb)
{
    for (int ij= 0; ij< iDl; ++ij)
        iMas[ij]= StrToInt(Is_Mas->Cells[iStolb][ij + 1]);
} //Vvod_Mas

//Осуществляет линейную сортировку
void Sort_Massiva(int_Mas &iMas, int iDl)
{
    for (int ij= 0; ij< (iDl - 1); ++ij)
        for (int iCan= (ij + 1), iByf; iCan< iDl; ++iCan)
            if (iMas[ij]> iMas[iCan])
            {
                iByf= iMas[ij];
                iMas[ij]= iMas[iCan];
                iMas[iCan]= iByf;
            } //if
} //Sort_Massiva

//Осуществляет суммирование массива
float Sym_Mas(int_Mas iMas, int iDl)
{
    float fS= 0;
    for (int ij= 0; ij< iDl; ++ij)
        fS += iMas[ij];
    return fS; //Оператор возврата результата
              //в точку ввода функции
```

```

    }// Sym_Mas

    //Выводит массив в таблицу
    void Vuvod_Mas(TStringGrid *Rez_Mas, int iRez_Massiv[],
                  int iDl, int iStolb)
    {
        for (int ij= 0; ij< iDl; ++ij)
            Rez_Mas->Cells[iStolb][ij + 1]= iRez_Massiv[ij];
    }//Vuvod_Mas

    void __fastcall TIsp_Polz_Fynkc::FormCreate(TObject *Sender)
    {
        //Длина более длинного массива m_Mas
        const int m= StrToInt(Dl_m_Mas->Text);
        //Определяем число строк таблиц
        Isx_Mas->RowCount= m + 1;
        Sort_Mas->RowCount= m + 1;
        //Называем столбцы таблиц
        Isx_Mas->Cells[0][0]= "№";
        Isx_Mas->Cells[1][0]= "n_Mas";
        Isx_Mas->Cells[2][0]= "m_Mas";
        Sort_Mas->Cells[0][0]= "№";
        Sort_Mas->Cells[1][0]= "n_Mas";
        Sort_Mas->Cells[2][0]= "m_Mas";

        //Нумерация строк таблицы
        for (int ij= 1; ij<= m; ++ij)
        {
            Isx_Mas->Cells[0][ij]= ij;
            Sort_Mas->Cells[0][ij]= ij;
        }//for_ij
    }//FormCreate

    void __fastcall TIsp_Polz_Fynkc::Dl_m_MasChange(TObject *Sender)
    {
        //Длина более длинного массива m_Mas
        const int m= StrToInt(Dl_m_Mas->Text);
        //Определяем число строк таблиц
        Isx_Mas->RowCount= m + 1;
        Sort_Mas->RowCount= m + 1;

        //Нумерация строк и очистка ячеек таблицы
        for (int ij= 1; ij<= m; ++ij)
        {
            Isx_Mas->Cells[0][ij]= ij;
            Isx_Mas->Cells[1][ij]= "";
            Isx_Mas->Cells[2][ij]= "";

            Sort_Mas->Cells[0][ij]= ij;
            Sort_Mas->Cells[1][ij]= "";
            Sort_Mas->Cells[2][ij]= "";
        }//for_ij
    }//Dl_m_MasChange

    void __fastcall TIsp_Polz_Fynkc::PyskClick(TObject *Sender)
    {
        const int n= StrToInt(Dl_n_Mas->Text),
            m= StrToInt(Dl_m_Mas->Text);
    }

```

```

int_Mas in_Mas, im_Mas;// Исходные массивы

//Заносим исходные данные в массивы
Vvod_Mas(Isx_Mas, in_Mas, n, 1);
Vvod_Mas(Isx_Mas, im_Mas, m, 2);

//Преобразование массивов
Sort_Massiva(in_Mas, n);
Sort_Massiva(im_Mas, m);

//Вывод массивов
Vuvod_Mas(Sort_Mas, in_Mas, n, 1);
Vuvod_Mas(Sort_Mas, im_Mas, m, 2);
float fS= Sym_Mas(in_Mas, n) + Sym_Mas(im_Mas, m);
Symma_Symm->Caption= "Сумма сумм массивов равна " +
                    FloatToStrF(fS, ffFixed, 8, 0);
} //PyskClick

```

Функции *FormCreate* и *Dl_m_MasChange* предназначены для установки числа строк обеих *Таблиц Строк* по числу элементов наибольшего массива *m_Mas* и нумерации их строк. Функция *FormCreate* выводит также названия столбцов *Таблиц Строк*.

Функция *Vvod_Mas(TStringGrid *Is_Mas, int_Mas & iMas, int iDl, int iStolb)* осуществляет ввод элементов массива из *Таблицы Строк* в переменную-массив. Поскольку *Таблица Строк* – визуальный объект, поэтому в функцию передаётся формальный параметр *Is_Mas*, который является указателем на тип *TstringGrid*. Как видите, имя формального параметра *Is_Mas* не тождественно имени *Таблицы Строк Isx_Mas* со значениями элементов исходных массивов. Имя этой таблицы выступает фактическим параметром при вызове функции *Vvod_Mas* в функции *PyskClick*. В теле функции *Vvod_Mas* обращение к свойствам объекта *Is_Mas* осуществляется через знак «->».

Вторым формальным параметром (*iMas*) функции *Vvod_Mas* является ссылка на тип *int_Mas*, который представляет собой тип одномерного целочисленного массива с числом элементов *nn*= 100. Этот параметр передаётся по ссылке, потому в теле функции его используют без каких-либо специальных знаков. Передача этого параметра – это один из способов передачи массивов в функцию, назовём его для определённости первым способом.

Целочисленные формальные параметры *iDl* и *iStolb* передаются в функцию по значению, поскольку не предусматривается их изменение в теле функции. Они описывают соответственно длину массива и номер столбца в *Таблице Строк*, из которой читаются значения конкретного массива. При вызове функции *Vvod_Mas* в функции *PyskClick* эти параметры для массива *n_Mas* заменяются фактическими параметрами *n* и 1, для массива *m_Mas* – параметрами *m* и 2 (см. рис. 8.1). Как видите, фактические параметры являются константами, что вполне допустимо для параметров-значений.

Функция *Sort_Massiva(int_Mas & iMas, int iDl)* осуществляет линейную сортировку одномерного массива. Формальный параметр *iMas* является ссылкой на тип *int_Mas*, а параметр *iDl* передаётся по значению. Метод линейной сортировки описан в разделе 6.2.

Функция *Sym_Mas(int_Mas iMas, int iDlina)* иллюстрирует ещё один способ передачи массива в функцию (назовём его вторым способом). Синтаксис передачи параметра *iMas* соответствует передаче параметра по значению. Эта функция осуществляет *суммирование* элементов массива, поэтому сами значения элементов *не изменяются*, и их не следует возвращать глобальной переменной. Однако на самом деле значения элементов *возвращаются* в глобальные переменные функции *PyskClick*.

Для проверки этого утверждения уберите знак амперсанта в заголовке функции *Sort_Massiva(int_Mas iMas, int iDl)* и оставьте неизменным её тело. При вызове такой модернизированной функции отсортированные элементы массива передаются в глобальную переменную (это обнаружится на интерфейсе после нажатия кнопки *Пуск*). Дело в том, что как отмечалось в разделе 8.4, массивы *невозможно* передать по значению. Они передаются либо по ссылке, либо по адресу, т.е. всегда являются параметром-переменной. По умолчанию (в случае отсутствия амперсанта и звёздочки) массив передаётся как константный указатель на неконстантные данные (см. п. 8.4.5).

Функция *Vuvod_Mas(TStringGrid *Rez_Mas, int iRez_Massiv[], int iDl, int iStolb)* иллюстрирует *третий* способ передачи массива в качестве параметра-переменной функции. Он заключается в том, что впереди формального параметра имени массива (*iRez_Massiv*) указывается тип элементов массива (*int*), а после имени массива ставятся пустые квадратные скобки: *int iRez_Massiv[]*. При этом вызов упомянутой функции (в функции *PyskClick*) с фактическим параметром осуществляется без квадратных скобок: *Vuvod_Massiva(Sort_Mas, in_Mas, n, 1)*;

В функции *PyskClick* решаются такие задачи.

1. Объявляются массивы *in_Mas, im_Mas*, заносятся в них значения соответственно из первого и второго столбцов таблицы *Isx_Mas* при помощи двух последовательных вызовов функции *Vvod_Mas*.
2. С использованием двух вызовов функции *Sort_Massiva* осуществляется их сортировка.
3. Вывод изменённых массивов выполнен при помощи двух последовательных вызовов функции *Vuvod_Mas*.
4. Дважды применяя функцию *Sym_Mas*, получена искомая сумма сумм исходных массивов.

Если поставленную задачу выполнять без применения пользовательских функций, то код приложения будет длиннее, менее понятен и малоприменим для использования в других приложениях.

Тело функции *PyskClick* в результате применения пользовательских функций оказалось очень компактным, оно состоит из последовательного вызова пользовательских функций. Для обработки двух массивов дважды вызывались функции *Vvod_Mas, Sort_Massiva* и *Vuvod_Mas*. При отказе от использования пользовательских функций программа станет существенно длиннее и менее понятной. Если в пользовательских функциях отказаться от применения формальных параметров, то тогда необходимо для *каждой* переменной *in_Mas* и *im_Mas* писать свои функции ввода, вывода, сортировки и суммирования, что весьма неразумно.

Если бы по условию задачи требовалось обработать всего лишь *один* массив, то в этом случае можно, в принципе, отказаться от применения формальных параметров. Однако возможность удобной последующей модернизации программы будет упущена. Если при этом отказаться и от применения пользовательских функций, то существенно ухудшится структурированность приложения. Приведенные рассуждения подтверждают необходимость использования пользовательских функций и их формальных параметров.

Напоминаем, что одни и те же пользовательские функции предназначены для обработки массивов с *различным* числом компонент. Именно поэтому в учебном приложении формальный параметр называется *iMas*, а фактические параметры – *in_Mas* и *im_Mas*, формальный параметр имеет имя *iDl*, а фактические параметры – *n* и *m*. Это делает программы более понятными и поэтому является определенным стандартом разработки программ: при описании пользовательских функций весьма желательно, чтобы идентификаторы формальных и фактических параметров функций не совпадали между собой. При несоблюдении этого требования разработчик программы может, например, ошибочно полагать, что локальная переменная является глобальной, что функция предназначена только для обработки конкретной переменной.

Необходимо отметить, что использование параметров-переменных в функциях, *возвращающих* любой тип данных кроме типа *void*, в принципе, допустимо, однако является *очень плохим* стилем программирования. В этом случае вызов функций иногда приводит к трудно контролируемым последствиям. Следует стремиться к тому, чтобы упомянутая функция возвращала *только одно* значение (т.е. в ней не использовались параметры-переменные). Эта рекомендация не относится к переменным-массивам, поскольку массивы *всегда* передаются в функцию только по адресу.

Рассмотренную учебную задачу используем для иллюстрации *других* способов передачи массивов в функцию. Наиболее наглядно их демонстрировать на функции *Sort_Massiva*, которая чётко диагностирует, что значения исходных массивов сортируются (т.е. изменяются), следовательно, передаются в функцию как параметр-переменная. С этой целью приведенный выше вариант функции *Sort_Massiva* следует последовательно заменять предлагаемыми ниже вариантами и делать отмечаемые ниже изменения в функции *PyskClick*.

Итак, четвёртый вариант передачи массива в функцию:

```
void Sort_Massiva(int_Mas *iMas, int iDl)
{
    for (int ij= 0; ij< (iDl - 1); ++ij)
        for (int iCan= ij+1, iByf; iCan<iDl; ++iCan)
            if ((*iMas)[ij]> (*iMas)[iCan])
            {
                iByf= (*iMas)[ij];
                (*iMas)[ij]= (*iMas)[iCan];
                (*iMas)[iCan]= iByf;
            }//if
}// Sort_Massiva
```

В этом варианте массив *iMas* передаётся в функцию как указатель на тип

int_Mas. Для осуществления в теле функции доступа к объекту переменной используется операция разыменовывание (звёздочка ставится слева от имени массива). Поскольку массив – это переменная не простого, а структурированного типа, то приходится использовать круглые скобки для указания того, что разыменовывание относится *только* к имени массива, индексы в квадратных скобках позволяют передвигаться от нулевого элемента массива (его адрес совпадает с именем массива) к любой выбранной ячейке – элементу массива. В случае применения упомянутого способа передачи массива в функцию при вызове такой функции перед фактическим параметром (именем массива) *следует ставить* знака амперсанта:

```
Sort_Massiva(&in_Mas, n);
Sort_Massiva(&im_Mas, m);
```

Пятый вариант передачи массива в функцию был показан в п. 8.6.6. Продемонстрируем его ещё раз на функции *Sort_Massiva*:

```
void Sort_Massiva(int *iMas, int iDlina)
{
    for (int ij= 0; ij< iDlina-1; ++ij)
        for (int iCan= ij + 1, iByf; iCan<iDl; ++iCan)
            if (iMas[ij]> iMas[iCan])
            {
                iByf= iMas[ij];
                iMas[ij]= iMas[iCan];
                iMas[iCan]= iByf;
            }//if
}// Sort_Massiva
```

Здесь имя массива объявляется указателем на простой тип *int* (тип элементов массива), а в теле функции от адреса нулевой ячейки массива (или имени массива) перемещаются к любым выбранным ячейкам при помощи индексов в квадратных скобках. Вызов этого варианта функции производится *без* знака амперсанта в её заголовке перед фактическим параметром-переменной:

```
Sort_Massiva(in_Mas, n);
Sort_Massiva(im_Mas, m);
```

Вопросы для самоконтроля

- ☐ Какие цели достигаются при использовании пользовательских функций?
- ☐ Укажите случаи, когда типы формальных и фактических параметров функций могут не совпадать между собой.
- ☐ По какой причине с использованием большого числа параметров-значений, например, типа *double*, пользовательские функции работают дольше, чем в случае применения параметров-переменных?
- ☐ Почему фактическим параметром пользовательских функций с параметром-переменной не может быть константа или математическое выражение?
- ☐ Чем обусловлена следующая рекомендация: в функциях, возвращающих любые типы данных за исключением *void*, нежелательно использовать параметры-переменные?
- ☐ Могут ли имена формальных и фактических параметров пользовательс-

ких функций совпадать между собой?

- ☐ Что нужно сделать для того, чтобы запретить доступ к пользовательским функциям из тех модулей приложения, в которых они не описаны?
- ☐ Назовите четыре способа возврата из пользовательских функций в точку их вызова.
- ☐ Назовите операцию, которая позволяет сделать доступной (видимой) в теле пользовательской функции глобальную переменную, одноимённую локальной переменной этой функции.
- ☐ Какой тип возвращает пользовательская функция, если при её описании не указан тип возвращаемого значения?
- ☐ Когда необходимо использовать прототипы функций?
- ☐ Где необходимо располагать параметры по умолчанию в заголовке пользовательских функций?
- ☐ В каких случаях в заголовке пользовательских функций ставится многоточие?
- ☐ Перечислите достоинства и недостатки применения параметров-переменных и параметров-значений.
- ☐ Нужно ли при вызове функции использовать знак амперсанта перед ссылочным фактическим параметром?
- ☐ Почему при вызове функции не используется знак взятия адреса перед фактическим параметром-указателем, являющимся визуальным объектом интерфейса?
- ☐ В каких случаях используется спецификатор *const* в заголовках пользовательских функций?
- ☐ Какие цели преследуются при перегрузке функций?
- ☐ В каких случаях можно применить шаблон семейства функций? Могут ли при описании шаблонной функции использоваться нешаблонные типы данных?

8.8. Задача для программирования

Задача 8.1. В матрице $M(4,3)$ найти сумму элементов третьей строки и сумму элементов второго столбца, отсортировать по возрастанию элементов упомянутые строку и столбец. Задачу решить с применением пользовательских функций.

8.9. Вариант решения задачи

Интерфейс приложения приведен на рис. 8.2. Текст файла реализации показан ниже.

Файл реализации.

```
const int nn= 100; //Максимальное число строк и столбцов
//Определение пользовательских типов
typedef float fMas[nn];
typedef float fMatr[nn][nn];
//Определение глобальных массивов
```

Задача 8.1

Число строк: 4 Число столбцов: 3 Номер строки: 3 Номер столбца: 2

Исходная матрица

	ст. 1	ст. 2	ст. 3
стр. 1	1	4	1
стр. 2	1	3	1
стр. 3	3	2	1
стр. 4	1	1	1

Результаты сортировки

№	стр. 3	ст. 2
1	1.00	1.00
2	2.00	2.00
3	3.00	3.00
4		4.00

Сумма элементов 3 строки равна 6.00
Сумма элементов 2 столбца равна 10.00

Пуск

Рис. 8.2. Интерфейс решения
Задача 8.1

```
fMas fSt_Mas, fCl_Mas; //Одномерные массивы
fMatr fM; //Матрица

//Заносит данные из таблицы в матрицу и массивы
void Chtenie(TStringGrid *Tabl,
            fMatr & fMa, int iStr, int iCl,
            fMas & fMas_Str, fMas & fMas_Cl,
            int inStr, int inCl)
{
    for (int ii= 1; ii<= iStr; ++ii)
        for (int ij= 1; ij<= iCl; ++ij)
        {
            //Заносим данные из таблицы в матрицу
            fMa[ii - 1][ij - 1]=
                StrToInt(Tabl->Cells[ij][ii]);

            //Переписываем строку
            fMas_Str[ij - 1]= fMa[inStr - 1][ij - 1];
            //Переписываем столбец
            fMas_Cl[ii - 1]= fMa[ii - 1][inCl - 1];
        }
    // for_ij
}

//Осуществляет линейную сортировку
void Sort_Mas(fMas & fMass, int iDl)
{
    for (int ij= 0; ij< iDl - 1; ++ij)
        for (int iCan= ij+1; iCan< iDl; ++iCan)
            if (fMass[ij]> fMass[iCan])
            {
                float fByf= fMass[ij];
                fMass[ij]= fMass[iCan];
                fMass[iCan]= fByf;
            }
    //if
}

//Осуществляет суммирование элементов массива
float Sym_Mas(float fMass[], int iDl)
{
    float fS= 0;
    for (int ij= 0; ij< iDl; ++ij)
        fS += fMass[ij];
}
```



```
    return fS;//Возврат результата в точку вызова
} // Symmirov_Massiva

void Vuvod_Mas(TStringGrid *Tab_Rez, int iCl,
               fMas fMass, int iDl)
{
    for (int ii= 1; ii<= iDl; ++ii)
        //Заносим данные из массива fMass
        // в столбец iCl таблицы Tab_Rez
        Tab_Rez->Cells[iCl][ii]=
            FloatToStrF(fMass[ii - 1], ffFixed, 8, 2);
} // Vuvod_Mas

void __fastcall TTZadacha_8_1::PyskClick(TObject *Sender)
{
    const int in_Str= StrToInt(n_Str->Text),
              in_Cl= StrToInt(n_Cl->Text),
              // Выбор номеров строки и столбца
              inStr= StrToInt(nom_Str->Text),
              inCl= StrToInt(nom_Cl->Text);
    Chtenie(Tab, fM, in_Str, in_Cl, fSt_Mas, fCl_Mas,
            inStr, inCl);
    // Установка числа строк таблицы Tabl_Rezylt
    // и нумерация её строк
    if (in_Str> in_Cl)
    {
        Tabl_Rez->RowCount= in_Str + 1;
        for (int ii= 1; ii<= in_Str; ++ii)
            Tabl_Rez->Cells[0][ii]= ii;
    } //if
    else
    {
        Tabl_Rez->RowCount= in_Cl + 1;
        for (int ij= 1; ij<= in_Cl; ++ij)
            Tabl_Rez->Cells[0][ij]= ij;
    } //else
    //Называем столбцы таблицы Tabl_Rezylt
    Tabl_Rez->Cells[0][0]= " №";
    Tabl_Rez->Cells[1][0]= "стр. " + IntToStr(inStr);
    Tabl_Rez->Cells[2][0]= "ст. " + IntToStr(inCl);
    float fS;
    fS= Sym_Mas(fSt_Mas, in_Cl);
    Sym_Stroki->Caption=
        "Сумма элементов " + IntToStr(inStr) +
        " строки равна " + FloatToStrF(fS, ffFixed, 10, 2);
    fS= Sym_Mas(fCl_Mas, in_Str);
    Sym_Stolbca->Caption=
        "Сумма элементов " + IntToStr(inCl) +
        " столбца равна " + FloatToStrF(fS, ffFixed, 10, 2);
    Sort_Mas(fSt_Mas, in_Cl);
    Sort_Mas(fCl_Mas, in_Str);

    Vuvod_Mas(Tabl_Rez, 1, fSt_Mas, in_Cl);
    Vuvod_Mas(Tabl_Rez, 2, fCl_Mas, in_Str);
} //PyskClick
```

```

void __fastcall TTZadacha_8_1::FormCreate(TObject *Sender)
{
    const int in_Str= StrToInt(n_Str->Text),
              in_Cl= StrToInt(n_Cl->Text),
              inom_Str= StrToInt(nom_Str->Text),
              inom_Cl= StrToInt(nom_Cl->Text);
    // Установка числа строк и столбцов таблицы Tab
    Tab->RowCount= in_Str + 1;
    Tab->ColCount= in_Cl + 1;
    // Установка числа строк таблицы Tabl_Rez
    // и нумерация её строк
    if (in_Str> in_Cl)
    {
        Tabl_Rez->RowCount= in_Str + 1;
        for (int ii= 1; ii<= in_Str; ++ii)
            Tabl_Rez->Cells[0][ii]= ii;
    }//if
    else
    {
        Tabl_Rez->RowCount= in_Cl + 1;
        for (int ij= 1; ij<= in_Cl; ++ij)
            Tabl_Rez->Cells[0][ij]= ij;
    }// else
    // Называем строки таблицы Tab
    for (int ii= 1; ii<= in_Str; ++ii)
        Tab->Cells[0][ii]="стр. " + IntToStr(ii);
    // Называем столбцы таблицы Tab
    for (int ij= 1; ij<= in_Cl; ++ij)
        Tab->Cells[ij][0]="ст. " + IntToStr(ij);
    // Называем столбцы таблицы Tabl_Rez
    Tabl_Rez->Cells[0][0]= " №";
    Tabl_Rez->Cells[1][0]= "стр. " + IntToStr(inom_Str);
    Tabl_Rez->Cells[2][0]= "ст. " + IntToStr(inom_Cl);
}//FormCreate

void __fastcall TTZadacha_8_1::n_StrChange(TObject *Sender)
{
    const int in_Str= StrToInt(n_Str->Text),
              in_Cl= StrToInt(n_Cl->Text);
    // Установка числа строк таблицы Tab
    Tab->RowCount= in_Str + 1;
    for (int ii= 1; ii<= in_Str; ++ii)
    {
        // Называем строки таблицы Tab
        Tab->Cells[0][ii]="стр. " + IntToStr(ii);
        // Очистка ячеек Tab
        for (int ij= 1; ij<= in_Cl; ++ij)
            Tab->Cells[ij][ii]= "";
    }//for_ii
    // Установка числа строк таблицы Tabl_Rez,
    // нумерация её строк и очистка ячеек
    if (in_Str> in_Cl)
    {
        Tabl_Rez->RowCount= in_Str + 1;
        for (int ii= 1; ii<= in_Str; ++ii)
        {

```




Урок 9. Файлы

9.1. Назначения файлов

9.1.1. Устройства для долговременного хранения информации. История вопроса

Как уже упоминалось на предшествующих уроках, память бывает оперативная и долговременная (постоянная). Скорость доступа к оперативной памяти очень высока (этим объясняется её название), однако после отключения электропитания все регенеративные схемы компьютера замирают, и информация в оперативной памяти пропадает.

Поэтому ещё на заре эры *электронных вычислительных машин* (ЭВМ) разработаны устройства для *постоянного* хранения информации. В те годы только одна ЭВМ обычно занимала целый этаж небольшого здания или помещение размером со спортзал. Обслуживанием этих сакраментальных машин занимались группы очень надменных и ужасно важных системных инженеров и операторов. Даже в настоящее время почти все системные программисты частично остаются такими же. А в те годы это были лица очень уж похожие на величественных фараонов. Как правило, все они носили специальные тапочки и белые халаты, очень загадочно смотрелись на фоне целого ряда больших железных шкафов, переливающихся разноцветными огнями своих индикаторов. Под фальшь-полом упомянутые шкафы с электроникой соединялись десятками кабелей, часто бронированных. Доступ программистов к ЭВМ был строго ограничен. Обычные программисты могли видеть чудо тогдашней техники только в маленькое окошечко, через которое сдавали колоды своих программ.

Колода программы представляла собой стопку продырявленных перфокарт – тонких картонок, на которых при помощи специальных прямоугольных отверстий-окошечек записывался (набивался) код программы и данные для обработки. Имелись специальные «читалки», предназначенные для облегчения чтения этой информации в восьмеричной системе счисления. Некоторые асы умели читать такие перфокарты и без читалок. Часто при набивке данных встречались ошибки, ведь эта операция осуществлялась вручную на специальных перфораторах. В этом случае *вся* перфокарта изготовлялась заново, она «перебивалась», или «неправильная» дырка заклеивалась, а новое окошечко в нужном месте прорезалось лезвием.

В семидесятые – восьмидесятые годы прошлого столетия колода с задачей в

лучшем случае «ставилась» на ЭВМ один раз в сутки. Получив распечатку с ошибкой, требовалось внести исправления в колоду (отнести в перфораторную заказ, проверить правильность набивки, вновь собрать колоду) и сдать её в упомянутое окошко. Затем наступали томительные сутки ожиданий. Конечно, некоторые программисты имели более свободный доступ к ЭВМ, однако для большинства он был весьма ограничен. Пожалуй, достаточно вспоминать этот кошмар (или оживлять ностальгию о былом).

Важно то, что в те годы перфокарты играли роль запоминающего устройства, используемого для *постоянного* хранения информации. При помощи перфокарт информация вводилась и выводилась из ЭВМ, на них хранились программы и массивы данных (исходные и результирующие).

Позже, в дополнение к перфокартам, появились *перфоленты* и *магнитные ленты*. Эти новшества воспринимались, как чудо. Вы просто не представляете радости тогдашних программистов и их руководителей, когда у них появилась возможность просто взять огромный массив данных (коробку с магнитной лентой) и унести. «Зачем и куда?» – поинтересуется наш читатель. А куда угодно: на другую, более мощную машину, заказчикам, расположенным в другом городе (мы возили их Москву и Ленинград), для последующего анализа при помощи иной программы – всего не перечесать.

Одним из наиболее важных преимуществ магнитных лент перед перфокартами и перфолентами являлось то, что они позволяли проводить *многочасовые* научно-технические эксперименты в *реальном* масштабе времени: давали возможность *оперативно* записывать огромные массивы данных. До широкого внедрения магнитных лент ответственный исполнитель научно-исследовательской темы важно наддувал щёки, когда ему удавалось «достать» тогдашнее чудо – перфоратор ПЛ-150 (ПЛ – перфоратор ленточный), ведь он позволял выводить 150 строк в секунду (тогда это было очень, очень много). Приходилось мириться с тем, что эти перфоленты часто рвались, а запись информации на них производилась столь медленно.

С появлением гибких и жёстких дисков ситуация изменилась радикально. Персональный компьютер *окончательно* поработил своих владельцев, часто пыливый пользователь «слазит» с него лишь после двенадцатичасовой работы, когда его голова уже затрудняется ясно соображать.

После такого небольшого экскурса в романтическое прошлое надеемся, что читатели наглядно представили те преимущества, которыми обладают пользователи современных персоналок, получив *в полное* распоряжение *все* ресурсы компьютера.

9.1.2. Что называется файлом

Любая информация, хранимая на постоянном запоминающем устройстве (гибком или жёстком диске, флеш-памяти, *CD* или *DVD*-дисках), называется файлом. Каждый файл характеризуется именем, что позволяет выполнять поиск конкретного файла среди других файлов диска. В переводе с английского слово

file означает *хранить информацию*, а также *картотека*, *досье* – все упомянутые значения верно характеризуют и новый смысл этого слова. Последнее время файлами также называют прозрачные чехлы, предназначенные для бережного и удобного сохранения документов. Файл в информационных технологиях – это поименованная область памяти на внешнем носителе, предназначенная для хранения данных, он может размещаться на гибких, жестких, *CD* или *DVD*- дисках, во флеш-памяти.

Имя файла в *MS-DOS* может содержать до 8 разрешенных символов. Чтобы использовать больше литер, информирующих о содержании файла (типе данных, какая программа его создала), обычно применяется расширение имени файла – последовательность до трех разрешенных символов. В интернет-технологиях имеются расширения до четырёх литер. Помимо информационной нагрузки расширения файлов позволяют открывать эти файлы соответствующими приложениями. Разрешенными символами являются прописные и строчные латинские буквы, арабские цифры и знак подчеркивания «_». В названиях файлов также допустимы следующие символы: «! @ # \$ % ^ & () ' ~ –», однако они меньше подходят для указанной цели. Имя файла может начинаться с любого разрешенного символа.

В *MS-Windows* имена файлов могут быть длинными, содержать пробелы и буквы кириллицы, в них различаются строчные и прописные буквы. Однако *настоятельно* рекомендуем не злоупотреблять этими возможностями: применять *только* латиницу, цифры и знак подчёркивания, ограничиваться именами файлов до 10 – 12 литер. В этом случае значительно меньше проблем возникнет с переносимостью и совместимостью файлов. Даже имена каталогов следует набирать *только* с использованием латинских букв. В противном случае *Builder* донимает сообщением о несуществующей ошибке.

Часто файлы упорядочены – распределены по каталогам и подкаталогам (папкам и вложенным папкам). Если программа, предназначенная для обработки файла, не находится с ним в одном и том же каталоге, то для обращения к файлу следует указывать его *полное* имя (имя файла с путём к нему). Файлы с исходными данными и результатами удобнее (на этапе разработки) располагать в текущем каталоге вместе с другими файлами проекта.

9.1.3. Что содержится в файлах

В файлах может находиться числовая информация, записанная с выхода измерительного устройства либо вручную, с использованием какого-либо текстового редактора. При проведении научно-технических экспериментов часто поток информации столь велик, что его не удаётся обработать сразу, «в реальном времени». В этом случае прибегают к записи данных в файл для их последующего детального анализа.

Имеются также файлы, содержащие базы данных, представляющие собой таблицы, в столбцы (называемые полями) которых заносится однородная информация. Например, данные о сотрудниках учреждения содержат фамилии, имена, от-

чества, даты рождения, адреса местожительства, должности, оклады и др.

Файлы могут играть роль связывающего звена между *разными* программами. Например, одна программа результаты своей работы сохраняет в файле, другая – данные этого файла распечатывает на принтере (в удобное время), третья – строит графики зависимостей (или производит иную обработку данных).

На предыдущих уроках во всех учебных проектах данные вводили при помощи клавиатуры, а выводили – на экран дисплея. На этом и всех последующих уроках используется иной порядок. Вся исходная информация заносится в файл. Программа *самостоятельно* обращается к файлу с исходными данными, обрабатывает их, а результат анализа выводит в отдельный файл с результатами. Такой порядок работы очень удобен при подготовке и анализе данных, ускоряет процесс отладки программ: в файл исходные данные вводятся только *один* раз.

9.1.4. Типы файлов

Существует два типа файлов:

- ☐ Двоичные или бинарные.
- ☐ Текстовые.

Данные в бинарных файлах (символы и числа) представлены в двоичной системе счисления, в машинных кодах. Такие файлы ещё называют файлами *непосредственного* или *произвольного доступа*, поскольку любой их элемент можно *очень быстро* найти, прочитать или записать в него новую информацию. Здесь имеется некоторая аналогия с массивами.

Элементами текстовых файлов также могут быть символы и числа, однако трактуются они как наборы *символов* в виде строк (длины которых могут быть различными). Для чтения (или записи) любой компоненты текстового файла необходимо вначале прочитать (просмотреть) все предшествующие элементы: элемент за элементом, строку за строкой. Поэтому текстовые файлы называются файлами *последовательного доступа*. Последовательный доступ осуществляется значительно медленнее непосредственного доступа.

9.1.5. Работа с файлами на физическом уровне

Работа с файлами состоит в том, что из них читают и в них записывают данные. Файлы – это информация, предназначенная для *долговременного* хранения, поэтому они располагаются на *периферийных* устройствах компьютера. Обращение к ним всегда было и остаётся *очень* медленным (по сравнению с доступом к данным, расположенным в оперативной памяти). Для сокращения числа обращений к долговременной памяти компьютера данные считываются и записываются не по одному биту или байту, а порциями. При этом используется специальный буфер.

При *чтении* информации из файла упомянутый буфер *автоматически* полностью заполняется данными, поэтому программа считывает данные не непосредственно из файла, а из буфера. Если программе требуется считать из файла

всего несколько байт, то буфер всё равно заполняется полностью. По мере чтения данных буфер опорожняется, как только он оказывается пустым, вырабатывается сигнал для передачи новой порции данных в буфер.

При *записи* в файл, данные вначале накапливаются в буфере, после его полного заполнения они «выталкиваются» (записываются, переносятся) в файл, затем в буфер вновь поступают данные и после его полного наполнения происходит следующее перемещение новой порции данных из буфера в файл и так далее до конца данных.

Последняя порция данных может не заполнить буфер полностью, поэтому для её «проталкивания» в файл придуманы специальные функции. Эти функции используются для *обновления файла* – перенесения содержимого буфера в файл, в том числе и последней неполной его порции. Но самым радикальным способом обновления файла является его *закрытие*. Эта операция не только разрывает канал связи с файлом, но и переносит последнюю *неполную* порцию в файл. Поэтому при записи в файл она является *обязательной*. В противном случае теряется часть данных или все данные, если их объём меньше объёма буфера.

Существует *определённый порядок* в работе с файлами. Вначале файл необходимо открыть, с тем, чтобы разрешить *потоку* информации (или данных) считываться или записываться в него (или то и другое). Как отмечалось, после завершения работы с файлом, его необходимо закрыть: ликвидировать канал связи с файлом или *прервать поток*. *Входной, выходной поток данных; прервать, возобновить поток, направление потока* – всё это термины *C++*, которые далее постараемся избегать, поскольку в *Builder* без них можно обойтись.

Столь детальная работа с файлами (иногда говорят – на *физическом уровне*) описана лишь для того, чтобы читатели представляли назначение функций, обеспечивающих упомянутые процессы. Такая работа с файлами присуща всем языкам программирования, разработанным и для *MS-DOS*, и *MS-Windows*. Однако если в *C* и *C++* программист вынужден иметь дело с целой серией функций, предназначенных для работы с файлами, разбираться во всех упомянутых потоках, то в *Builder* всё значительно проще и удобнее.

В *Builder* работать с файлами можно в стиле *C* и *C++*, однако функции, предназначенные для реализации этих архаических возможностей, в настоящем пособии не рассматриваются, поскольку в изучаемой среде программирования имеются *значительно* более удобные *новые* функции, являющиеся прекрасным инструментарием разработки приложений.

Такое отношение к старым функциям оправдано также и тем, что на самом деле большинство из них в *Builder* *остались* и честно *работают*. Например, заботу об открытии и закрытии текстовых файлов взяли на себя соответственно конструктор и деструктор классов (разговор о них ведётся на четырнадцатом уроке): файлы открываются и закрываются при помощи вызова *старых* испытанных функций. Просто эта необходимая работа осталась *за кадром*, чтобы появилась возможность эффективно воплощать программистские идеи *в самом кадре*.

Новые классы в *Builder* также впитали всё лучшее, что было создано в ходе развития языка *C++*, и любезно предоставляют *удобные* механизмы для реализа-

ции всех завоеваний своего родителя. Предшествующее повествование приведено лишь для *самого общего* представления о файлах и не более. Сейчас, наконец, дадим ответ на, видимо наиболее болезненный, вопрос: «Как же работать с этими файлами?». Вначале познакомимся с текстовыми файлами.

9.2. Простые приложения с текстовыми файлами

Открыв текстовый файл текстовым редактором легко разобраться с его содержанием, поскольку элементами таких файлов являются все символы клавиатуры (в частности, буквы и цифры). Таким образом, текстовые файлы хранят свои данные в очень удобном для человека виде, их удобно просматривать, проверять и анализировать.

Бинарные же файлы представлены в машинных кодах, и поэтому их содержание для человека окажется понятным только при помощи специальных программ обработки. После открытия двоичного файла текстовым редактором, увидите лишь набор бессмысленных символов: треугольников, ромбиков, сердечек, мордочек, символов псевдографики и др. О бинарных файлах рассказывается в разделе 9.4. Сейчас же приступим к изучению текстовых файлов.

9.2.1. Многострочные окна редактирования Мето и RichEdit

Для начала рассмотрим следующую очень простую задачу: вывести содержание файла *Isxodn_f.txt* на интерфейс приложения в два равных по габаритам окна; после редактирования текста во втором (правом) окне записать его в новый файл *Rezylt_f.rtf*. На рис. 9.1. показан интерфейс разрабатываемого приложения.

Для решения этой задачи в каталоге учебного проекта создайте текстовый файл с именем *Isxodn_f.txt* и запишите в него несколько строк произвольного текста. Расширение файла может быть любым допустимым расширением, либо отсутствовать вовсе. Если файл создан в файловой оболочке *FAR*, то кириллица воспроизводится неверно, при наборе текста с использованием приложений *Про-*

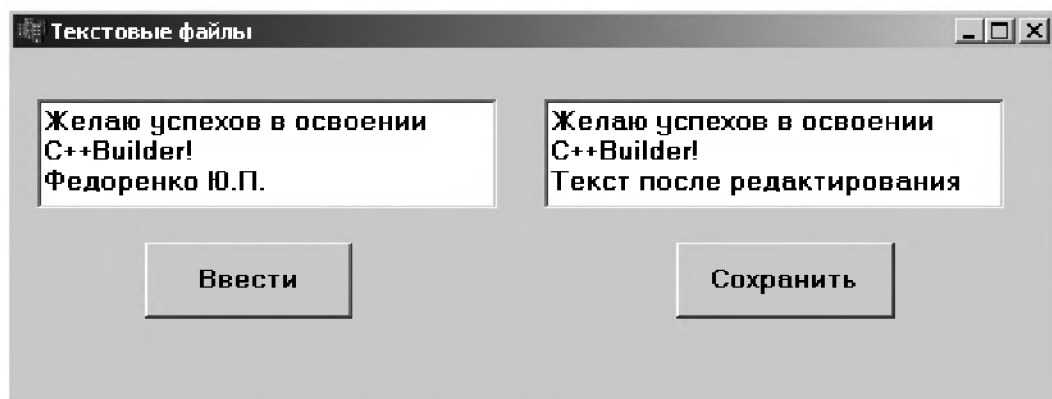


Рис. 9.1. Интерфейс приложения *Текстовые файлы*

водник, Мой Компьютер и Builder символы национальных алфавитов отображаются правильно (причина искажений разъясняется на двенадцатом уроке). В Builder 5.0 текстовый файл порождается при помощи команды *File/New.../Text*, а в Builder 6.0 – командой *File/New...Other/Text*.

Имена файлов с *исходными* и *результатирующими* данными могут использоваться в теле приложения многократно. Поэтому в начале программы удобно заменить их лаконичными синонимами или *файловыми переменными*. В этом случае при выборе вместо них других имён файлов в программе изменения вносятся *только один* раз. Имена файлов могут содержать достаточно много литер и включать длинные пути к ним в файловой системе компьютера, поэтому упомянутый приём делает тело приложения более лаконичным и ясным. Файловые переменные – это набор символов, поэтому они имеют текстовый тип *String*.

Объявите в файле реализации приложения файловые переменные *Isx_f* и *Rez_f*, соответствующие файлам с исходными *Isxodn_f.txt* и изменёнными *Rezylt_f.rtf* данными:



```
String Isx_f = "Isxodn_f.txt", //Объявление файловых
Rez_f = "Rezylt_f.rtf"; // переменных
```

Если файл с данными *не находится* в одном и том же каталоге со всеми файлами проекта, то перед его именем указывается путь к нему, например:

```
String Isx_f = "D:\\Katalog\\PodKatalog\\Isxodn_f.txt";
```

Обращаем внимание на то, что в программе левый слеш в пути к файлу должен быть *удвоенным*.

Познакомимся с новыми визуальными объектами типа *TMemo* и *TRichEdit* – многострочными окнами редактирования. В окнах типа *TMemo* выбранные атрибуты форматирования относятся *ко всему* тексту, а в окнах типа *TRichEdit* *произвольному фрагменту* текста можно программно установить *свои* атрибуты форматирования (механизмы такого форматирования рассматривать не будем). Заготовки (компоненты) этих объектов находятся соответственно на вкладках

Standard и *Win32*, они выглядят вот так:  (*Memo*)  (*RichEdit*). Породите упомянутые объекты на форме, имена объектов *Memo1* и *RichEdit1*, предложенные *Builder* по умолчанию, оставьте без изменения. Вертикальные габариты этих объектов рекомендуется увеличить, чтобы видеть выведенными из файла *Isxodn_f.txt* все его строки.

В данном примере эти компоненты используются для ввода (из файла на экран) и вывода (с экрана в файл или в переменные приложения) *многострочного* текста. Свойство *Lines* (строки) этих объектов позволяет хранить и показывать в окнах интерфейса строки, загруженные из текстового файла с использованием функции *LoadFromFile*. Содержимое свойства *Lines* при помощи функции *SaveToFile* можно записать в текстовый файл.

Вывод текста (из файла *Isx_f*) в упомянутые выше окна редактирования осуществляют следующие две строки кода:

```
Memol->Lines->LoadFromFile(Isx_f); //Загрузка данных из файла
RichEdit1->Lines->LoadFromFile(Isx_f);
```

Эти строки вставьте в функцию обработки нажатия кнопки *Ввести* (см. рис. 9.1). Свойство *Lines* объектов *Memo1* и *RichEdit1* принадлежит классу *TStrings* (или имеет тип *TStrings*). Функция (метод) *LoadFromFile* этого класса позволяет выводить (загружать) текстовые данные из файла в окно редактирования, а функция (метод) *SaveToFile* осуществляет копирование (сохранение) строк текста из окна редактирования в текстовый файл (заданный файловой переменной). Поэтому в функцию обработки нажатия кнопки *Сохранить* вставьте такой код:

```
RichEdit1->Lines->SaveToFile(Rez_f); //сохранение данных
```

Содержимое файла *Rezylt_f.rtf* с текстом из окна *RichEdit1* можно посмотреть только в текстовом редакторе *MS-Word* и других редакторах, поддерживающих формат *RTF* (*Rich Text Format* – обогащённый текстовый формат). При этом использованное расширение *rtf* не имеет значения (выбрано нами лишь в информационных целях), расширение допустимо не использовать или заменить, например на *txt*. Если же сохранение выполнять из окна *Memo1*, то файл *Rezylt_f.rtf* открывается без искажений *любым* текстовым редактором, поскольку его формат соответствует простому текстовому формату:

```
Memol->Lines->SaveToFile(Rez_f);
```

Необходимо заметить, что в объектах типа *TMemo* и *TRichEdit*, также как и в объектах типа *TEdit*, предусмотрены стандартные комбинации «горячих» клавиш: *Ctrl+C* (или *Ctrl+Ins*), *Ctrl+X* – соответственно копирование и вырезание выделенного текста в буфер обмена; *Ctrl+V* (или *Shift+Ins*) – вставка текста из буфера; *Ctrl+Z* – отмена последней команды редактирования.

В классе *TStrings* имеется также метод *Strings*, осуществляющий *доступ* к любой строке введенного текста объектов типов *TMemo* и *TRichEdit* (в *Инспекторе Объектов* это свойство *Strings*). Например, для последовательного вывода первых трёх строк из объекта *Memo1* можно привести следующие операторы:

```
ShowMessage(Memol->Lines->Strings[0]); //Желаю успехов в освоении  
ShowMessage(Memol->Lines->Strings[1]); //C++Builder  
ShowMessage(Memol->Lines->Strings[2]); //Федоренко Ю.П.
```

Как видно из этого примера, метод (свойство) *Strings* хранит строки текста из *Memo1* (или *RichEdit1*) в одномерном массиве, индексы которого начинаются с нуля.

В свойстве *Lines* имеется подсвойство *Count*, в котором хранится *текущее* число строк, загруженных из файла в окно (или набранных в окне) объекта типа *TMemo* или типа *TrichEdit*. Это подсвойство доступно программно и *только для чтения*. С его использованием очень просто решается вопрос о том, сколько же строк текста загружено из файла в конкретном случае:

```
ShowMessage("Загружено " + IntToStr(Memol->Lines->Count) + " строк");
```

9.2.2. Невизуальный объект типа *TStringList*

Интерфейс приложения, в котором иллюстрируется применение нового объекта, показан на рис. 9.2. При его запуске функция *FormCreate* осуществит вывод данных из файла в окно объекта *Memo1*, кнопка *Выход* (с именем *Vuxod*) –

объект типа *TButton*, осуществляет закрытие приложения.

Для чтения из файла и записи в файл текстовых данных удобно использовать *невизуальный* объект типа *TStringList*. Также как и объекты типов *TMemo* и *TRichEdit*, объект типа *TStringList* хранит загруженный в него текст в виде *набора строк*, количество записанных строк автоматически указывается в свойстве *Count*, а доступ к произвольной строке осуществляется при помощи свойства *Strings*. Для порождения указанного невидимого объекта (глобального) в начале файла реализации объявите указатель на тип *TStringList* и осуществите его инициализацию при помощи операции *new*:

```
TStringList *Spisok = new TStringList;
```

Здесь *Spisok* – это имя порождаемого объекта типа *TStringList* (более правильно говорить – имя указателя на тип *TStringList*). Перед загрузкой текста из файла в объекты типов *TMemo*, *TRichEdit*, *TStringList* необходимо вначале выяснить существует ли требуемый файл на диске. Если файл не найден, то следует вывести соответствующее сообщение об этом. Для корректной работы с файлами создадим функцию *Zagryzka*, которой поручим выполнять эту важную работу.

```
void Zagryzka (TStringList *Nabor_Strok, String Isxod_f)
{
    try //Аналогично для TMemo и TRichEdit
    {
        Nabor_Strok->LoadFromFile(Isxod_f);
    } //try
    catch (...)
    {
        ShowMessage("Файл \" " + Isxod_f + " \" не найден");
        Abort();
    } //catch
} //Zagryzka
```

Если пользователь для выхода из приложения воспользуется граничной пиктограммой интерфейса, то приложение корректно освободит все ранее занятые им ресурсы, в том числе закроет все незакрытые файлы и уничтожит все объекты как визуальные, так и невидимые (конечно, речь идёт об указателях на объекты). Если же вооружить пользователя специальной кнопкой *Выход*, то програм-

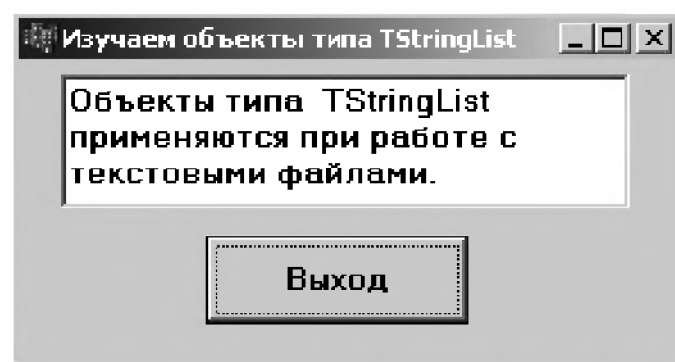


Рис. 9.2. Интерфейс приложения *Изучаем объекты типа TStringList*

мисту следует *самому* побеспокоиться об уничтожении порождённых в приложении динамических *невизуальных* объектов. С этой целью в функции *VuxodClick* перед именем объекта *Spisok* вызывается операция *delete*, предназначенная для освобождения памяти, занимаемой динамическим объектом:

```
void __fastcall TTekst_f::VuxodClick(TObject *Sender)
{
    delete Spisok;
    Close();
} //VuxodClick
```

Честно говоря, операцию *delete* в этой функции можно и не применять, поскольку функция *Close()*, предназначенная для корректного закрытия основной формы приложения, успешно справляется и с её работой. Функция *Close()* сама вызовет операцию *delete* для объекта *Spisok*, её работа эквивалентна щелчку по крайней справа граничной пиктограмме. Однако разработка функции *VuxodClick* приучает к хорошему стилю программирования: перед завершением программы уничтожать все динамические переменные и объекты, которые в ней порождались (или уничтожать динамические переменные и объекты сразу же после того, как в их существовании отпала необходимость). Трудяга *Builder*, который старается всё взять на себя, «мешает» придумать более подходящий лаконичный пример, иллюстрирующий применение этого правила.

В файле реализации введите глобальные файловые переменные:

```
String Isx_f = "Isxodn_f.txt",
      Rez_f = "Rezylt_f.txt";
```

Далее приведите описанные выше функции *Zagryzka* и *VuxodClick*. Тело функции *FormCreate*, в которой используется объект *Spisok* типа *TStringList* имеет вид:

```
Zagryzka(Spisok, Isx_f);
ShowMessage(Spisok->Strings[0]); //Объекты типа TStringList
ShowMessage(Spisok->Strings[1]); //применяются при работе
ShowMessage(Spisok->Strings[2]); //с текстовыми файлами
ShowMessage("Загружено " + IntToStr(Spisok->Count) + " строк");
Spisok->SaveToFile(Rez_f);
Memo1->Lines->LoadFromFile(Rez_f);
```

Вызов пользовательской функции *Zagryzka* осуществляет загрузку *всех* строк текста из файла *Isxodn_f.txt* в объект *Spisok*. Далее в окнах функции *ShowMessage* последовательно выводятся первые три строки этого текста и число строк, загруженных в объект *Spisok*. В предпоследней строке функции *FormCreate* производится копирование содержимого объекта *Spisok* во вновь создаваемый текстовый файл *Rezylt_f.txt*. Теперь в файлах *Isxodn_f.txt* и *Rezylt_f.txt* находятся одни и те же строки текста, они иллюстрируются в окне объекта *Memo1* при помощи последней строки кода функции *FormCreate*.



5
4
3
2
1

Рис. 9.3. Файл с исходным массивом



Исходный массив
5
4
3
2
1
Массив после сортировки
1
2
3
4
5

Рис. 9.4. Файл с результатами

9.3. Обработка массивов в текстовых файлах

9.3.1. Чтение из файла и запись в файл одномерного массива

Учебное приложение выполняет такие задачи:

1. Ввести в программу элементы одномерного массива из файла *Isxodn_f.txt* исходных данных (см. рис. 9.3), после чего вывести их в файл *Rezylt_f.txt* с результатами (см. рис. 9.4).
2. Отсортировать массив по возрастанию элементов.
3. Вывести упорядоченный массив в файл с результатами (см. рис. 9.4).

Перед запуском программы на выполнение создайте файл исходных данных – одномерный массив из n целых чисел (каждое число находится на отдельной строке файла). Файл исходных данных создаётся при помощи *любого* текстового редактора или в редакторе *Builder* с использованием команды: *File/New/Other/Text*. Интерфейс приложения показан на рис. 9.5.

В файле реализации приложения объявляются такие глобальные константы, типы, объекты и файловые переменные:

```
const int nn= 100; //Максимальная длина массива
typedef int int_Mas[nn]; //Определение пользовательского типа
TStringList *Spisok= new TStringList;
String Isx_f,
        Rez_f;
```

В приложении *Обработка одномерных массивов* после нажатия кнопки *Пуск* вызывается функция *PyskClick*. В этой функции вводится локальный одномерный массив *in_Mas*, конкретные имена файлов из полей ввода с именами *Isxodn_Fajl* и *Rezylt_Fajl* инициализируют глобальные файловые переменные *Isx_f* и *Rez_f*.

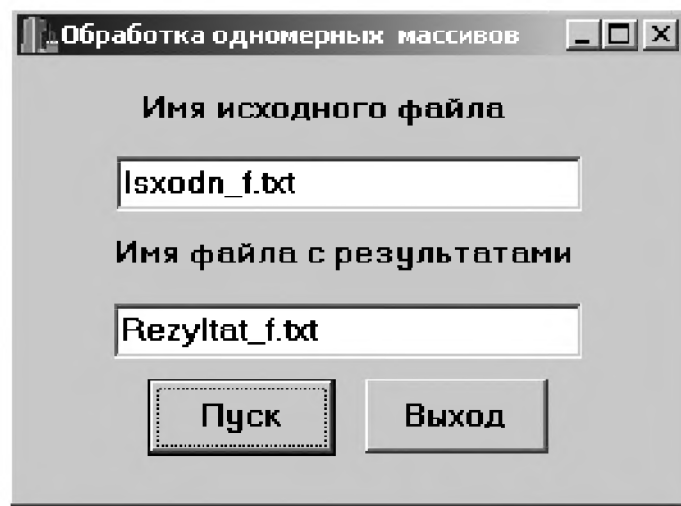


Рис. 9.5. Интерфейс приложения Обработка одномерных массивов

```
int_Mas in_Mas; //Объявление переменной-массива типа int_Mas
Isx_f= Isxodn_Fajl->Text; //Ввод имён файлов
Rez_f= Rezylt_Fajl->Text;
```

Далее с использованием стандартных и пользовательских функций решается поставленная задача. Вначале при помощи пользовательской функции *Zagryzka* (она описана в предшествующем разделе) происходит загрузка исходного одномерного массива из файла исходных данных *Isx_f* в объект типа *TStringList* с именем *Spisok*. Затем в функции *PyskClick* при помощи свойства *Count* объекта *Spisok* определяется число *n* элементов массива:

```
int n= Spisok->Count; //Число элементов массива
```

Каждый элемент массива находится на отдельной строке свойства *Strings* объекта *Spisok*. Это даёт возможность в пользовательской функции *Vvod_Mas* применить упомянутое свойство для перезаписи элементов массива из строк свойства в соответствующие ячейки одномерного массива *iMas* (имя формального параметра функции). В этой же пользовательской функции иллюстрируется применение метода *Clear()* объекта *Spisok* для очистки списка.

```
void Vvod_Mas(int_Mas & iMas, int iDlina)
{
    for (int ij= 0; ij< iDlina; ++ij)
        iMas[ij]= StrToInt(Spisok->Strings[ij]);
    Spisok->Clear(); //Очистка переменной Spisok
} //Vvod_Mas
```

После вызова этой функции с фактическими параметрами *in_Mas* и *n* объект *Spisok* окажется очищенным, а массив будет содержать числовые значения, скопированные в него из текстового файла. Запишем теперь в объект *Spisok* при помощи его метода *Add* заголовок *Исходный массив* (см. рис. 9.4):

```
Spisok->Add("Исходный массив");
```

Сортировка массива выполняется пользовательской функцией *Sort_Mas*:

```
void Sort_Mas(int_Mas & iMas, int iDlina)
//Осуществляем линейную сортировку
```

```

{
    for (int ij= 0; ij< iDlina-1; ++ij)
        for (int iCan= ij + 1, iByf; iCan<iDlina; ++iCan)
            if (iMas[ij]> iMas[iCan])
            {
                iByf= iMas[ij];
                iMas[ij]= iMas[iCan];
                iMas[iCan]= iByf;
            }//if
} // Sort_Mas

```

Пользовательская функция *Vuvod_Mas* предназначена для вывода элементов одномерного массива в объект *Spisok* для последующего его записи в файл с результатами *Rez_f*:

```

void Vuvod_Mas(int iMas[], int iDlina)
{
    for (int ij= 0; ij< iDlina; ++ij)
        Spisok->Add(iMas[ij]);
} //Vuvod_Mas

```

Каждый элемент выводимого массива последовательно записывается в отдельную строку объекта *Spisok* при помощи метода *Add*, с использованием этого же метода в функции *PyskClick* в файл с результатами выводятся комментарии к выводимым данным. После заполнения объекта *Spisok* (см. рис. 9.4) все его данные копируются в файл при помощи команды:

```
Spisok->SaveToFile(Rez_f);
```

В заключение приводим полное тело функции *PyskClick*:

```

int_Mas in_Mas; //Объявление массива типа int_Mas
Isx_f= Isxodn_Fajl->Text; //Ввод имён файлов
Rez_f= Rezylt_Fajl->Text;

Zagryzka(Spisok, Isx_f);
int n= Spisok->Count; //Число элементов массива

Vvod_Mas(in_Mas, n);
//Добавляем новую строку списка
Spisok->Add("Исходный массив");
Vuvod_Mas(in_Mas, n);
Sort_Mas(in_Mas, n);
Spisok->Add("Массив после сортировки");
Vuvod_Mas(in_Mas, n);
//Создаём файл Rez_f и записываем в него значения
// из переменной Spisok
Spisok->SaveToFile(Rez_f);

```

9.3.2. Чтение из файла и запись в файл двумерных массивов

Ввод из файла элементов двумерного массива в переменную-матрицу и вывод компонент матрицы после её обработки в текстовый файл проиллюстрируем на решении следующей задачи. Матрица $M(5,8)$ находится в файле *Isx_f.txt*, зане-

сти её в двумерный массив fM , отсортировать по возрастанию значений элементы третьей строки матрицы. Исходную и изменённую матрицы вывести в файл *Rez_f.txt*.

В предшествующем приложении было показано, как выбирать любую заданную строку из текста, расположенного в текстовом файле. Такой строкой может быть и строка матрицы чисел. Как вереницу чисел, разделённых пробелами (или пробелом) из текстового вида превратить в строку числовой матрицы? Для решения такой задачи предлагаем вначале эту текстовую строку типа *String* (назовём её *Stroka*) превратить в указатель на *char* – указатель на строку символов с нулевым символом в конце (назовём его *Yk_Char*). В объекте этого указателя будут храниться тот же набор символов, что и в переменной *Stroka*, но с нулевым символом в конце. Текстовые (или строковые) и символьные переменные изучаются лишь на двенадцатом уроке. Поэтому для осознанной работы с функциями разрабатываемого приложения приведём необходимые сведения о таких переменных.

Так вот, текстовые переменные типа *String* (или *AnsiString*) – это такие переменные, которые введены только в *C++Builder*. При объявлении в них записывается начальное (стартовое) значение в виде пустой строки (""). Длина таких переменных определяется фактической длиной текста, который записан в переменную такого типа. Для обработки переменных типа *String* в *C++Builder* имеются очень удобные функции. Текстовые переменные в *C++* заканчиваются нулевым символом, позволяющим определять их длину и диагностировать конечный символ строки.

Обработка переменных текстового типа в *C++* существенно упрощается, если обрабатывать не сами текстовые переменные, а указатели на них. Поскольку в случае указателей можно легко перемещаться вдоль вереницы символов такой переменной и выбирать любой из них (для чтения или записи).

Ниже познакомимся с функциями обработки как переменных типа *String*, так и указателей на *char*. Опишем также функции преобразования между указанными переменными. Вот так преобразуют переменную *Yk_Char* в переменную типа *String*:

```
String Stroka= String(Yk_Char);
```

Можно это же преобразование выполнить несколько по-иному:

```
String Stroka= (String) Yk_Char;
```

Такие варианты *явного* преобразования типа упоминались на предшествующем уроке. Для преобразования переменной типа *String* в переменную указатель на *char* используют функцию *c_str()*:

```
Yk_Char= Stroka.c_str();
```

Функция *strtok(Yk_Char, Stroka_Razdelitelej)* ищет первое вхождение разделителей перечисленных в строке символов *Stroka_Razdelitelej* и усекает при их появлении переменную *Yk_Char*. Усечённая часть переменной *Yk_Char* является возвращаемым значением функции *strtok*. Среди символов в строке *Stroka_Razdelitelej* могут быть, например, пробел, знак табуляции, точка, запятая, точка с запятой и др. Последующий (повторный) вызов этой функции с парамет-

рами *strtok(NULL, Stroka_Razdelitelej)* усекает оставшуюся часть строки по *следующему* разделителю. Таким образом, последовательный вызов функции *strtok* позволяет *расчлени*ть строку *Yk_Char* на части, находящиеся между разделительными символами. Согласно условию задачи строка разделителей состоит *только* из пробела, т.е. разделителями служат пробелы (один или более пробельных символов). Имя функции *strtok* включает сокращения двух слов: *string* (строка) и *token* (символ).

Интерфейсы разрабатываемого и предшествующего приложений отличаются только полем ввода номера строки матрицы, предназначенной для сортировки. После приведенных пояснений попытайтесь (с использованием комментариев в тексте) разобраться в коде приложения.

```
//Файл реализации
//Максимальное число строк и столбцов матрицы
const int inSt= 100, inCl= 100;
//Определение пользовательских типов
//Тип fMatr[inSt][inCl]: матрица чисел
//типа float с inSt числом строк и inCl столбцов
typedef float fMatr[inSt][inCl];
//Определяем глобальные переменные
//Приложение автоматически определяет число
//строк и столбцов матрицы, хранит их в следующих переменных
int in_Cl, in_St; //Для хранения текущих значений
fMatr fM; // Задание матрицы
TStringList *Spisok = new TStringList;
String Isx_f, Rez_f; //Файловые переменные

void Zagryzka(TStringList *Nabor_strok, String Isxod_f)
{
    try
    {
        Spisok->LoadFromFile(Isxod_f);
    } //try
    catch (...)
    {
        ShowMessage("Файл \" " + Isxod_f + " \" не найден");
        Abort();
    } //catch
} //Zagryzka

void Vvod_Stroki_Matr(TStringList *Nabor_Strok,
                     fMatr & fMa, int iSt, int & iCl)
// iSt- строка, выбранная для ввода
// iCl - число столбцов матрицы, определяется в теле
//настоящей функции
{
    //Объявляем две переменные-указатели на тип char
    char *Yk_Char, *Yk_Char_1;
    //Записываем одномерный массив в строку Stroka
    String Stroka= Nabor_Strok->Strings[iSt];
    //Преобразуем переменную Stroka типа String в
    //переменную-указатель на char с именем Yk_Char
    Yk_Char= Stroka.c_str();
```

```
String Stroka_1;//Для временного хранения фрагментов строки
// Усекаем строку после первого пробела
Yk_Char_1= strtok(Yk_Char, " ");
int ij;//Для подсчёта числа столбцов матрицы
//Если переменная не пуста, т. е. указатель указывает не на 0
if (Yk_Char_1)
{
    //Строку символов преобразуем в строку String
    Stroka_1= String(Yk_Char_1);
    ij= 0;//Очистка счётчика числа столбцов матрицы
    //Записываем значения нулевого элемента строки в матрицу
    fMa[iSt][ij]= StrToInt(Stroka_1);
}//if
//Усечение оставшейся части строки
// Stroka по очередному пробелу
while (Yk_Char_1)
{
    Yk_Char_1= strtok(NULL, " ");
    if (Yk_Char_1)
    {
        Stroka_1= String(Yk_Char_1);
        ++ij;
        fMa[iSt][ij]= StrToInt(Stroka_1);
    }//if
}//while
//Число столбцов записываем в глобальную переменную
//Передача параметра через параметр-переменную
iCl= ij;//Отсчитываются от 0
}//Vvod_Stroki_Matr

void Vvod_Vsex_Strok(TStringList *Nabor_Strok,
                    fMatr & fMa, int iSt)
//Число строк матрицы iSt определяется в функции PyskClick
{
    for (int ii= 0; ii< iSt; ++ii)
        Vvod_Stroki_Matr(Nabor_Strok, fMa, ii, in_Cl);
}//Vvod_Vsex_Strok

void Sortirovka(fMatr & fMa, int iCl, int iSt)
//Линейная сортировка выбранной строки iSt
{
    for (int ij= 0; ij< iCl; ++ij)
        for (int iCan= ij + 1; iCan<= iCl; ++iCan)
            if (fMa[iSt][ij]> fMa[iSt][iCan])
            {
                float fByf= fMa[iSt][ij];
                fMa[iSt][ij]= fMa[iSt][iCan];
                fMa[iSt][iCan]= fByf;
            }//if
}// Sortirovka

void Vuvod_Matr(fMatr fMa, int iSt, int iCl)
{
    for (int ii= 0; ii< iSt; ++ii)//Отсчёт от 1 в Count
    {
        String Stroka= "";//Очистка не обязательна
```

```

        for (int ij= 0; ij<= iCl; ++ij) //Отсчёт от 0
            Stroka +=FloatToStrF(fMa[ii][ij], ffFixed, 10, 1) + " ";
        Spisok->Add(Stroka);
    } //for_ii
} // Vuvod_Matr

void __fastcall TForm1::VuxodClick(TObject *Sender)
{
    delete Spisok;
    Close();
} //VuxodClick

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    //Ввод номера строки для сортировки
    const int inom_Stro= StrToInt(nom_Stroki->Text);
    Isx_f= Isxodn_Fajl->Text; //Ввод имён файлов
    Rez_f= Rezylt_Fajl->Text;
    Zagryzka(Spisok, Isx_f);
    in_St= Spisok->Count;
    Vvod_Vsex_Strok( Spisok, fM, in_St);
    Spisok->Clear();
    Spisok->Add("Исходный массив:");
    Vuvod_Matr(fM, in_St, in_Cl);
    Sortirovka(fM, in_Cl, inom_Stro - 1);
    Spisok->Add("Массив после сортировки");
    Vuvod_Matr(fM, in_St, in_Cl);
    Spisok->SaveToFile(Rez_f);
    ShowMessage("Файл с итоговыми результатами готов!");
} //PyskClick

```

Обращаем внимание на то, что в функции *Vuvod_Matr* заголовок цикла по строкам *ii* имеет вид:

```

for (int ii= 0; ii< iSt; ++ii),
а по столбцам ij выглядит так:
for (int ij= 0; ij<=iCl; ++ij).

```

В первом случае используется знак «<», а во втором – «<=». Это обусловлено тем обстоятельством, что свойство *Count* объекта *Spisok* возвращает *истинное* число строк введенного текста, в функции *Vvod_Stroki_Matr* число столбцов отсчитывается от нуля, в матрице *fMatr* номера строк и столбцов, как принято в C++, отсчитываются также от нуля. По указанной причине в функции *Sortirovka* заголовок внутреннего цикла имеет вид:

```

for (int iCan= ij + 1; iCan<= iCl; ++iCan)

```

Это приложение без какого-либо изменения можно использовать для обработки *одномерных* массивов, элементы которых записаны в файле в виде *одной строки*.

9.4. Двоичные файлы

Данные в двоичных файлах представляют собой последовательность двоичных цифр. В таких файлах нет каких-либо разделителей – пробелов, символов конца

строки и т. д. Поэтому двоичные файлы имеют существенно *меньший* объём, чем текстовые файлы с теми же данными. Достоинством бинарных файлов является также и то, что операции чтения и записи данных в них производятся *намного* быстрее, чем в текстовых. Это обусловлено тем, что в бинарных файлах отсутствует необходимость форматирования, перевода данных в текстовое представление и обратно. Быстродействие чтения-записи и малый объём данных – это немаловажные преимущества в сравнении с текстовыми файлами.

При просмотре двоичных файлов в каком-либо текстовом редакторе вереницы двоичных цифр (биты) группируются в байты, поэтому, как отмечалось в начале урока, содержимое таких файлов выглядит в виде набора бессмысленных литер (знаков). При отладке программы анализировать и просматривать результаты удобнее в текстовых файлах (которые являются частным случаем двоичных файлов). Если памяти и быстродействия современных компьютеров окажется недостаточно при решении конкретной задачи, чтобы работать с текстовыми файлами, то после отладки основных этапов программы можно прибегнуть к эксплуатации двоичных файлов, раз они такие «компактные» и «быстродействующие». Не очень пламенная любовь автора к таким файлам не должна передаваться читателям: двоичные файлы – весьма полезны и широко применяются при решении научных и инженерных задач. Кроме отмеченных достоинств, следует напомнить ещё одно: только в двоичных файлах имеется *свободный* доступ к данным для их чтения и записи в *произвольной* последовательности (как в массивах). В текстовых же файлах производится *последовательная* обработка информации (строка за строкой), что приводит к уменьшению быстродействия приложений.

Как отмечалось в начале урока, потоки, широко применяемые в C и C++, рассматривать не будем. Но в C++ *Builder*, имеются ещё два варианта работы с двоичными файлами. Первый из них связан со структурами. Этот тип данных изучается лишь на тринадцатом уроке. Поэтому сейчас ограничимся рассмотрением только второго варианта, основанного на применении дескрипторов (*handle*).

Что собой представляет такой дескриптор? В операционной системе имеется специальная таблица, в которую заносится информация о каждом файле. Строка таблицы, в которой приводятся все параметры файла, помечается целым числом (номером). Этот номер, назначаемый системой при порождении файла, называют дескриптором, в дальнейшем повествовании будем называть его просто *номером файла*.

Вспомните работу с файлами на физическом уровне. В C++ *Builder* при использовании двоичных файлов процессы открытия, закрытия файлов и буферизации данных в ходе этих процессов не скрываются, как в случае текстовых файлов. Эти и другие процессы выполняют специальные функции, с некоторыми из них познакомимся в последующих программах. Ограничимся лишь самыми важными на наш взгляд функциями с тем, чтобы проиллюстрировать только основные этапы работы с двоичными файлами и не дублировать справочник. Не беспокойтесь, если отдельные стороны их функционирования при первом чтении окажутся не совсем ясными.

Функция *FileCreate(String Imja_Fajla)* создаёт файл с именем *Imja_Fajla* и

возвращает его номер (значение типа *int*). Если файл создать не удалось (нет места на диске, файл с таким именем уже существует в заданном каталоге и др.), то возвращается -1 . Созданный файл считается *открытым файлом*.

Если файл уже существует (был создан до начала работы вашего приложения), то для *открытия* канала связи с ним в одном из существующих режимов, файл следует открыть с использованием функции *FileOpen(String Imja_Fajla, int Regim)*. Эта функция возвращает номер файла, либо -1 , если файл не удалось открыть (открыт другим приложением, нет на диске и др. причины). В таблице приведены три режима открытия файлов (формальный параметр *Regim* в функции *FileOpen*).

Имя константы	Значение	Режим открытия
<i>fmOpenRead</i>	0000	Только для чтения
<i>fmOpenWrite</i>	0001	Только для записи
<i>fmOpenReadWrite</i>	0002	Для чтения и записи

Рекомендуем применять символьные, а не числовые (они представлены в восьмеричной системе счисления) значения режимов. Имеются режимы, запрещающие другим приложениям читать и записывать в файл; режим полного доступа к файлу из других приложений.

Чтение данных из файла осуществляется при помощи функции *FileRead(int Nom, void *Perem, int Razm_Perem)*. Эта функция из файла с номером *Nom* выводит блок данных объёмом *Razm_Perem* байтов в переменную *Perem*, возвращает число прочитанных байтов или -1 , если чтение информации из файла оказалось невозможным.

Функция *FileWrite(int Nom, void *Perem, int Razm_Perem)* выполняет *запись* из переменной *Perem* в файл с номером *Nom* блок данных объёмом *Razm_Perem* байтов, возвращает число записанных байтов или -1 , если запись информации в файл не осуществилась.

При работе с файлами вводится понятие *указатель позиционирования* файла и *позиция указателя* файла. Если в файл не записаны *никакие* данные, следовательно, в нём находятся данные с объёмом в 0 байт. В этом случае говорят, что указатель файла указывает на нуль. По мере записи данных в файл позиция указателя *перемещается* на число байтов *записанного* блока данных. Имеется возможность указатель позиционирования установить в требуемое положение относительно начала, конца или текущего положения указателя файла. Таким образом, позицию указателя файла передвигается в требуемое положение, начиная с которого можно *считывать* или *записывать* данные.

Позиция указателя смещается функцией *FileSeek(int Nom, int Sdvig, int Nachalo)*, которая перемещает позицию указателя файла с номером *Nom* на блок данных объёмом *Sdvig* байтов, начиная с позиции *Nachalo*. Эта функция возвращает 0 при успешном завершении своей работы. Перевод слова *seek* – искать.

9.4.1. Простое приложение с двоичным файлом

После нажатия кнопки *Пуск* приложение создаёт двоичный файл с именем *Rezylt_f*, запишет в него из переменной *Sx_1* строку текста «*Запись в файл*». Далее этот файл закрывается и открывается вновь в режиме *только для чтения*. После этого содержимое файла *Rezylt_f* копируется (читается) в новую текстовую переменную *Sx_2*. Для наглядной иллюстрации работы упомянутых функций значение переменной *Sx_2* «*Запись в файл*» показывается в окне функции *ShowMessage*, аргумент этой функции класса *String* вызывает метод, необходимый для преобразования двоичных данных в текстовые. Изучите теперь тело функции *PyskClick*:

```
int iNom; //Номер файла
String Rez_f = "Rezylt_f"; //Файл с данными
iNom= FileCreate(Rez_f); //Создаём файл Rez_f
if (iNom== -1)
    ShowMessage("Файл не удалось создать");
//Объявляем переменную Sx_1 и записываем в неё данные
String Sx_1 = "Запись в файл";
//Копируем данные из переменной Sx_1 в файл с номером iNom
FileWrite(iNom, &Sx_1, sizeof(Sx_1));
//Закрываем файл Rez_f с номером iNom
FileClose(iNom);
String Sx_2; //Объявляем ещё одну переменную
//Открываем файл Rez_f в режиме только для чтения
iNom= FileOpen(Rez_f, fmOpenRead);
//Копируем из файла Rez_f данные в переменную Sx_2
FileRead(iNom, &Sx_2, sizeof(Sx_2));
//Выводим содержимое переменной Sx_2 в текстовом виде
ShowMessage(Sx_2);
```

9.4.2. Одномерный массив в двоичном файле

В этом приложении в функции *PyskClick* задаётся и инициализируется одномерный целочисленный массив *iMas* из *n* элементов (интерфейс приложения не приводится, поскольку на нём имеется *только* кнопка *Пуск*), функция *FileCreate* создаёт (в текущей директории) двоичный файл *Dvoichn_f*, а функция *FileWrite* записывает в него содержимое массива *iMas*. Далее данные из этого файла копируются (читаются) в одномерный массив *iMas_1*. Однако такая запись осуществляется не сразу.

Для знакомства с работой *новых* функций файл *Dvoichn_f* вначале закрывается функцией *FileClose*, а затем открывается функцией *FileOpen* в режиме *Только для чтения*. После открытия файла *Dvoichn_f* в таком режиме его содержимое читается в массив *iMas_1*, однотипный с массивом *iMas*. После этого производится «склеивание» через пробел значений компонент массива *iMas_1* в виде текстовой строки *Stroka*. С использованием объекта *Spisok* типа *TStringList* эта строка выводится в текстовый файл с именем *Tekstov_f*. Если вы откроете этот файл, то увидите там элементы массива *iMas_1*, совпадающими с компонентами исходно-

го массива *iMas*. Ниже приводится тело функции *PyskClick*.

```
const int n= 5;//Длина массива
int iMas_1[n], iMas[n]= {5, 4, 3, 2, 1};
int iNom_Dv_F;
String F_Dvoich= "Dvoichn_f",
        F_Tekst = "Tekstov_f";
TStringList *Spisok = new TStringList;
iNom_Dv_F= FileCreate(F_Dvoich);
if (iNom_Dv_F== -1)
{
    ShowMessage("Файл не удалось создать");
    Abort();
} //if
FileWrite(iNom_Dv_F, &iMas, sizeof(iMas));
FileClose(iNom_Dv_F);
iNom_Dv_F= FileOpen(F_Dvoich, fmOpenRead);
FileRead(iNom_Dv_F, iMas_1, sizeof(iMas_1));
String Stroka= "";
for (int ij= 0; ij< n; ++ij)
    Stroka += IntToStr(iMas_1[ij]) + " ";
Spisok->Add(Stroka);
Spisok->SaveToFile(F_Tekst);
Spisok->Clear();
```

9.4.3. Матрица в двоичном файле

Для приложения *Матрица в двоичном файле* также как и в предшествующем приложении не приводится его интерфейс, поскольку на нём присутствует *только* кнопка *Пуск*. Двумерный целочисленный массив *iMatr* размером $n \cdot n$ задаётся в функции *PyskClick* при помощи функций *randomize* и *random* таким образом, чтобы его компоненты находились в диапазоне от 0 до $n \cdot n$ (от 0 до 25). Далее порождается двоичный файл *F_Dvoich*, в него записывается матрица *iMatr* сгенерированных ранее данных. Затем файл закрывается и открывается вновь в режиме *Только для чтения*, после чего содержимое файла *Dvoichn_f* копируется в матрицу *iMatr_1*, однотипную с матрицей *iMatr*. Для перевода двоичных данных этой матрицы к текстовому виду содержимое последней матрицы построчно записывается в объект *Spisok* типа *TStringList*, после чего все строки этого объекта переписываются в текстовый файл *Tekstov_f*, где их можно смотреть и анализировать.

При выводе элементов матрицы *iMatr_1* в объект *Spisok* использовано форматирование её элементов: для *однозначных* компонент пробел добавляется перед и за компонентом, для *двузначных* компонент пробел добавляется только позади компонента. В этом случае столбцы матрицы в файле *Tekstov_f* не искривлены, представляют собой строго вертикальный набор чисел.

```
const int n= 5;
randomize();
int iMatr[n][n], iMatr_1[n][n];
for (int ii= 0; ii< n; ++ii)
    for (int ij= 0; ij< n; ++ij)
        iMatr[ii][ij]= random(n*n+1);
```



```

int iNom_Dv_F;
String F_Dvoich= "Dvoichn_f",
        F_Tekst = "Tekstov_f";
iNom_Dv_F= FileCreate(F_Dvoich);
if (iNom_Dv_F== -1)
{
    ShowMessage("Файл не удалось создать");
    Abort();
}
FileWrite(iNom_Dv_F, &iMatr, sizeof(iMatr));
FileClose(iNom_Dv_F);
iNom_Dv_F= FileOpen(F_Dvoich, fmOpenRead);
FileRead(iNom_Dv_F, iMatr_1, sizeof(iMatr_1));
TStringList *Spisok = new TStringList;
for (int ii= 0; ii< n; ++ii)
{
    String Stroka= "";
    //Форматированный вывод матрицы
    for (int ij= 0; ij< n; ++ij)
        if (iMatr_1[ii][ij]<= 9)
            Stroka += " " + IntToStr(iMatr_1[ii][ij]) + " ";
        else
            Stroka += IntToStr(iMatr_1[ii][ij]) + " ";
    Spisok->Add(Stroka);
}
Spisok->SaveToFile(F_Tekst);
Spisok->Clear();

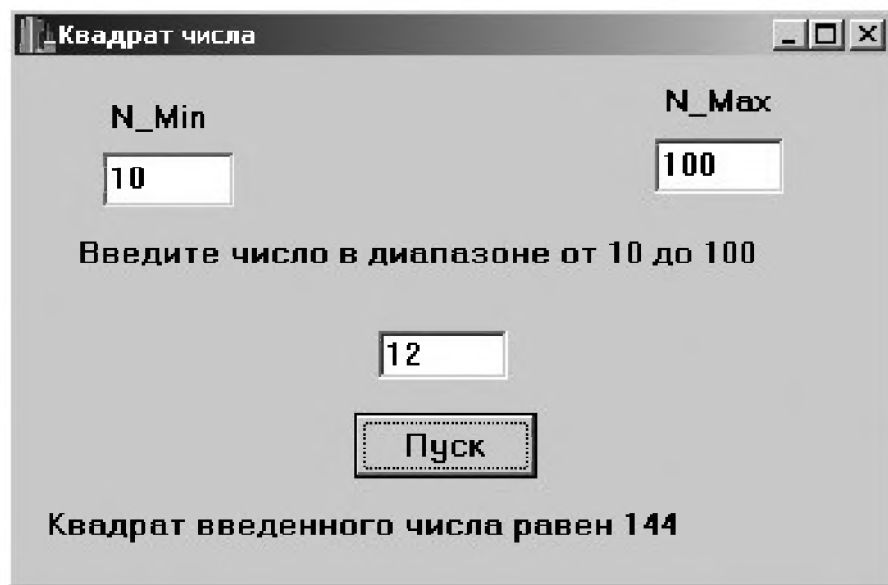
```

9.4.4. Приложение Квадрат числа

Интерфейс учебного приложения показан на рис. 9.6. Пользователь в поле ввода с именем *Vvod_Chisla* вводит число в заданном диапазоне целых чисел. После нажатия кнопки *Пуск* квадрат введенного числа появляется в поле вывода с именем *Rez*. Имена полей ввода границ диапазона называются соответственно *N_Min* и *N_Max*.

В этом приложении создается двоичный файл *Fajl_Kvadratov*, в него записываются квадраты чисел из введенного диапазона при помощи пользовательской функции *Fajl_Kvadr()*. В объект типа *TEdit* с именем *Vvod* пользователь вводит число. Значение квадрата этого числа (и всех других из заданного диапазона чисел) находится в файле *Fajl_Kvadratov*. Для того чтобы прочесть это значение (квадрат введенного числа), указатель файла передвигается в позицию, где расположено это значение. Это оказывается возможным в пользовательской функции *Slyga()* при помощи стандартной функции *FileSeek(iNom, (int)sizeof(float)*(iN - iN_Min), 0)* на основе значения числа *iN* и размера его типа *sizeof(float)*.

Для установки указателя файла в нужное положение его необходимо сместить от начала файла (0 байт) на *sizeof(float)*(iN - iN_Min)* байт. Поскольку в последнем выражении участвуют переменные типа *int* и *float*, то результирующее значение имеет тип *float*. Поэтому для согласования типа второго фактического

Рис. 9.6. Интерфейс приложения *Квадрат числа*

параметра функции *FileSeek* используется *явное* преобразование его типа к типу *int*: $(int)sizeof(float)*(iN - iN_Min)$. Дело в том, что в *Builder* функция *FileSeek* является перегруженной, поэтому для вызова *нужной* функции применяется явное преобразование типа второго параметра. Читатель, видимо понимает, что такая «сверхзадача» учебного приложения и путь её реализации были выбраны лишь для компактной иллюстрации работы изучаемых функций.

```
//Файл реализации приложения Квадрат числа.
...
#include <math.h>
String Rez_f;
const int n= 1000;//Максимальное число элементов массива
    int iN_Min;//Минимальное вводимое число
        iN_Max;//Максимальное вводимое число
float fMas[n];//Глобальный массив
String F_Kvadr = "Fajl_Kvadratov";
int iN;//Для запоминания введенного числа
float fR;//Для хранения квадрата введенного числа

void Fajl_Kvadr()//Создаёт двоичный файл, содержащий
    //квадраты чисел от N_Min до N_Max
{
    for (int ij = 0; ij< iN_Max - iN_Min; ++ij)
        fMas[ij]= pow(iN_Min + ij, 2);
    int iNom;
    iNom= FileCreate(F_Kvadr);
    if (iNom== -1)
    {
        ShowMessage("Файл не удалось создать");
        Abort();
    }//if
    FileWrite(iNom, &fMas, sizeof(fMas));
    FileClose(iNom);
}//Fajl_Kvadr

void Slyga()//Считывает задаваемое число в переменную iN и
```

```
//записывает квадрат его значения из файла в переменную fR
{
    if ((iN< iN_Min) || (iN> iN_Max))
    {
        ShowMessage("Вы ввели число вне запрашиваемого диапазона");
        Abort();
    } //if
    int iNom;
    iNom= FileOpen(F_Kvadr, fmOpenRead);
    FileSeek(iNom, (int) sizeof(float)*(iN - iN_Min), 0);
    FileRead(iNom, &fR, sizeof(fR));
    FileClose(iNom);
} // Slyga

void __fastcall TDvoich_f::FormCreate(TObject *Sender)
{
    iN_Min= StrToInt(N_Min->Text);
    iN_Max= StrToInt(N_Max->Text);
    Ykazanie->Caption= "Введите число в диапазоне от " +
        IntToStr(iN_Min) + " до " + IntToStr(iN_Max);
} //FormCreate

void __fastcall TDvoich_f::N_Min_Change(TObject *Sender)
{
    iN_Min= StrToInt(N_Min->Text);
    Ykazanie->Caption= "Введите число в диапазоне от " +
        IntToStr(iN_Min) + " до " + IntToStr(iN_Max);
} // N_MinChange

void __fastcall TDvoich_f::N_Max_Change(TObject *Sender)
{
    iN_Max= StrToInt(N_Max->Text);
    Ykazanie->Caption= "Введите число в диапазоне от " +
        IntToStr(iN_Min) + " до " + IntToStr(iN_Max);
} //N_MaxChange


void __fastcall TDvoich_f::PyskClick(TObject *Sender)
{
    //Запоминаем введенное число
    iN= StrToInt(Vvod->Text);
    Fajl_Kvadr();
    Slyga();
    Rez->Caption= "Квадрат введенного числа равен " +
        FloatToStrF(fR, ffFixed, 10, 0);
} //PyskClick
```

9.5. Улучаем интерфейс приложения

На интерфейсах всех приложений, выпускаемых различными фирмами, имеются пункты меню для ввода разнообразных команд. Большинство этих команд, как правило, дублируются горячими клавишами. Комбинация таких клавиш обычно приводится непосредственно в меню. Пользователи приложений привыкли к такой удобной интуитивно понятной организации интерфейса. Поэтому научимся порождать упомянутые средства оформления внешнего вида основного окна

программ. Полигоном для их освоения послужит приложение *Квадрат числа*. Модернизируем вначале это приложение таким образом, чтобы ввод пределов осуществлялся через пункт меню *Ввод пределов* и его вложенных пунктов *Ввод N_Min*, *Ввод N_Max*, а выход из программы выполнялся при помощи пункта меню *Выход*.

9.5.1. Создаём меню

Прежде всего, сделайте каталог *Izuch_Menu* и скопируйте в него все файлы приложения *Квадрат числа*. Затем, на вкладке *Standard Палитры Компонентов* найдите компонент *MainMenu*, он выглядит вот так: . Породите на форме объект типа *TMainMenu* с именем *MainMenu1* (имя по умолчанию). Не задумывайтесь о его размещении на форме: этот объект – невидимый, поэтому он останется *невидимым* при работе приложения, однако самостоятельно сгенерирует пункты меню с его подпунктами и расположит их в стандартном для них месте – под заголовком приложения.

Вызовите визуальный редактор *MainMenu* – дважды щёлкните по объекту *MainMenu1*, увидите окно, показанное на рис 9.7. Это наиболее удобный способ вызова визуального редактора. Во втором способе следует выделить объект *MainMenu1* на форме, затем перейти в *Инспектор Объектов*, выбрать свойство *Items* (пункты) и нажать кнопку с многоточием, расположенную рядом с этим свойством.

Нажатие клавиши *Enter* переносит курсор в поле ввода свойства *Caption текущего* пункта меню (это же можно осуществить щелчком мыши в упомянутом поле ввода). Введите в этом поле название первого пункта меню *Ввод пределов*. После ввода заголовка указанного пункта ещё раз нажмите клавишу *Enter* (либо щёлкните мышью по форме) – фокус ввода переместится на первый *подпункт* пункта меню *Ввод пределов* (см. рис. 9.8). В его свойстве *Caption* введите заголовок *Ввод N_Min*.

После *каждого* нажатия клавиши *Enter* генерируется заготовка *нового* подпункта. Заголовок второго (нижнего) подпункта сделайте таким: *Ввод N_Max* (см. рис. 9.9). Только *после* ввода заголовка *очередного* подпункта нажатие клавиши *Enter* приводит к появлению заготовки для *следующего* подпункта. Если заголовков подпункта не введен, то нажатие клавиши *Enter* осуществляет лишь после-



Рис. 9.7. Визуальный редактор *MainMenu*



Рис. 9.8. Ввод подпункта вложенного меню



Рис. 9.9. Вид всех подпунктов вложенного меню

довательные переключения между редактором меню и *Инспектором Объектов*. Для *уничтожения* любого подпункта меню его необходимо вначале выделить, а затем воспользоваться клавишей *Delete* или командой *Delete* контекстного (подручного) меню этого подпункта.

В свойстве *Caption* можно *редактировать* ранее введенные заголовки пунктов и подпунктов. Для вставки *нового* подпункта вложенного меню необходимо выделить тот подпункт, *впереди* (выше) которого осуществляется вставка, и нажать клавишу *Insert* или ввести эту же команду из подручного меню выделенного подпункта. Изменить *очерёдность* пунктов или подпунктов меню удобнее всего методом буксировки: выделенный пункт или подпункт перетаскивайте мышью на нужное место. Пункт можно сделать подпунктом и наоборот.

В приложениях часто близкие по содержанию подпункты меню группируют при помощи разделительных горизонтальных линий. Для размещения упомянутых линий *после* выбранного подпункта меню нажмите клавишу *Insert* и в поле ввода свойства *Caption* нового подпункта введите тире, после чего нажмите клавишу *Enter*.

Уровень вложений меню может быть *произвольным*. Однако не рекомендуем делать более трёх уровней вложенности. Ввод *подменю* какого-либо подпункта осуществляется так: выделите выбранный подпункт, затем в его контекстном меню введите команду *Create SubMenu* (создать подменю). Далее все операции подобны рассмотренным выше.

Новые пункты меню, расположенные *справа* за введенным пунктом, порождаются следующим образом: щёлкните мышью справа от *существующего* пункта в пустом прямоугольнике (см., например, рис. 9.8, 9.9) и повторите все указанные выше операции по порождению подпунктов (если они необходимы). На рис. 9.10 показано окно редактора после ввода второго пункта меню. В нашей учебной задаче вложенные разделы второго пункта меню не предусмотрены.

Вставка дополнительных пунктов между существующими пунктами меню или уничтожение пунктов осуществляется с использованием клавишам *Insert* и *Delete* либо вводом этих же команды из подручного меню.

Научимся теперь создавать заготовку функции для обработки нажатия конкретного пункта меню. Если в редакторе меню выделен, например, последний пункт *Выход*, то выполните по нему *двойной* щелчок. Если же редактор меню свёрнут, тогда щёлкните *один* раз по упомянутому пункту меню *на форме* приложения. В теле функции-заготовки укажите функцию закрытия формы *Close()*.



Рис. 9.10. Ввод второго пункта меню

Название по умолчанию этой функции – *N2Click* (Щелчок по второму пункту меню). Ранее были введены только *заголовки* пунктов и их подпунктов, а имена этих объектов, присвоенные *Builder* по умолчанию, пока не изменялись. Это не поздно сделать и сейчас. Пункт *Выход* в *Инспекторе Объектов* именууйте *Vuxod* (после чего имя функции *N2Click* изменится на *VuxodClick*), пункту *Ввод пределов* – дайте имя *Vvod_Predelov*. Подпункты *Ввод N_Min* и *Ввод N_Max* пусть соответственно называются *NMin* и *NMax*.

Далее займёмся пунктом *Ввод пределов*. Сделаем так, чтобы поля ввода значений *N_Min* и *N_Max* (с такими же именами), а также их заголовки (имена последних объектов соответственно *Vvod_N_Min* и *Vvod_N_Max*) при запуске приложения были бы *невидимыми*, появлялись бы только при вызове соответствующего подпункта меню. С этой целью, всем перечисленным выше объектам, в *Инспекторе Объектов* в свойстве *Visible* установите значение *false*. В тело заготовок функций по обработке нажатий упомянутых подпунктов вставьте операторы, которые в свойстве *Visible* названных объектов (вместо изначального значения *false*) установят *true*. Вот так, например, будет выглядеть функция *NMinClick*:

```
void __fastcall TKvadrat_Chisla::NMinClick(TObject *Sender)
{
    Vvod_N_Min->Visible= true;
    N_Min->Visible= true;
} // NMinClick
```

После ввода значений пределов соответствующие поля ввода и их заголовки должны *автоматически исчезать* с интерфейса, такой режим работы очень удобен, он реализован в большинстве приложений ведущих фирм. Кроме того, потеря фокуса активности (ввода) должна использоваться для *обновления* значения предела, выбранного для редактирования (прежнее значение заменяется новым, текущим). Все перечисленные задачи решаются функцией по обработке события *OnExit*. Приведём эту функцию, например, для поля ввода *N_Min*.

```
void __fastcall TKvadrat_Chisla::N_MinExit(TObject *Sender)
{
    iN_Min= StrToInt(N_Min->Text);
    Vvod_N_Min->Visible= false;
    N_Min->Visible= false;
} // N_MinExit
```



Рис. 9.11. Назначение горячих клавиш для подменю

Напоминаем, ввод команд меню (также как и нажатие командных кнопок) можно выполнить при помощи клавиатуры: нажать клавишу *Alt*, а затем, не отпуская её, щёлкнуть по клавише с подчёркнутой буквой в заголовке меню. Подчёркивание осуществляется размещением знака амперсанта в заголовке пункта или подпункта меню (в свойстве *Caption*) перед буквой, которую требуется выделить. В пункте *Ввод пределов* подчеркните строчную букву «в» в слове *Ввод*, в пункте *Выход* – букву «ы».


9.5.2. Создаём горячие клавиши


Свойство *Shortcut* визуального редактора *MainMenu* определяет *клавиши быстрого доступа* к подпунктам меню или *горячие клавиши*. С их применением пользователь, даже не заходя в меню, может выполнить операции, предусмотренные конкретными его подпунктами. Для назначения клавиш быстрого доступа в раскрывающемся списке свойства *Shortcut* выберите желаемую комбинацию клавиш. Эта комбинация появится в строке подменю (см. рис. 9.11, обратите внимание на подчёркнутые буквы в названии пунктов, введенные в п. 9.5.1). Регистр «горячей» буквы не имеет значения. Горячие клавиши можно назначить только подпунктам меню, но не его пунктам (при последнем уровне вложенности). При выборе комбинаций горячих клавиш:

- ☐ командам, одноимённым с командами популярных приложений (например, *Word*, *Excel* и др.), рекомендуется назначать общепринятые комбинации клавиш (например, *Сохранить* – *Ctrl+S*);
- ☐ общепринятые (уже занятые) комбинации клавиш не рекомендуется назначать пунктам меню, если они не совпадают по назначению с командами большинства приложений.

9.5.3. Создаём кнопки быстрого доступа


Как и в большинстве приложений *Windows* под строкой горизонтального (главного) меню создадим панель управления с кнопками, нажатие которых дублирует команды меню. С этой целью на вкладке *Палитры Компонентов Win32*

дважды щёлкните по компоненту *ToolBar* . Объект *ToolBar1* (имя по умолчанию) автоматически разместится под строкой меню, а не в центре формы, как было бы с другим объектом. Этот объект «знает» своё местожительство, оно *всегда* находится *под* строкой меню. Объект *ToolBar1* представляет собой пока лишь *пустую* панель управления, которую *можно* заполнить кнопками быстрого доступа.

Для решения этой задачи разместим на панели вначале *только один* экземпляр типа *TSpeedButton* (быстрая кнопка). Заготовка этого объекта находится на закладке *Additional Палитры Компонентов* и выглядит вот так . Перед двойным щелчком по компоненту *SpeedButton* объект *ToolBar1* *необходимо* выделить, тогда объект *SpeedButton1* (имя по умолчанию) *автоматически* расположится в начале панели управления. В противном случае, после появления на форме объекта *SpeedButton1* вне объекта *ToolBar1*, его *невозможно* отбуксировать *внутрь* объекта *ToolBar1*.

У первой быстрой кнопки пока нет *никакого* изображения, которое бы, отличало её от других кнопок. Изображение или картинку можно нарисовать при помощи специального графического редактора, подобрать готовый рисунок из библиотеки, поставляемой вместе с *Builder*, или модернизировать одну из картинок из этой библиотеки – сделать из неё интуитивно понятную пиктограмму, подсказывающую назначение конкретной команды, вызываемой нажатием кнопки.

Выбор *готовой* картинки и её размещение (загрузка) на поверхности кнопки осуществляются так:

- ❑ Выделите объект *SpeedButton1*, перейдите в *Инспектор Объектов*, выберите свойство *Glyph* (рельефно выделенный знак или символ) и щёлкните по кнопке с троеточием, находящейся справа от этого свойства . В результате появится окно *Редактора Картинок Picture Editor* (см. рис. 9.12). Далее щёлкните по кнопке *Load* (загрузить), затем в папке с именем *Buttons* выберите одну из 162 картинок (их вид иллюстрируется в специальном окне). Эта папка находится по адресу: *C:\Program File\Common File\Borland Shared\Images\Buttons*.

- ❑ Нажмите кнопку *Открыть*, затем – *ОК*.

Все стандартные картинки состоят из *двух* частей: изображение кнопки в нажатом и отжатом состоянии. Для каждой кнопки можно создать *всплывающую подсказку*, появляющуюся (если для свойства *ShowHint* (показать подсказку) выбрано значение *true*) при наведении мыши на эту кнопку. Такие подсказки записываются в свойстве *Hints* (в *Инспекторе Объектов* либо программно).

Имеется два способа заставить кнопку выполнять работу подпункта или пункта меню. Наиболее разумно назначить обработку события *OnClick* для выбранной кнопки уже *существующей* функции, предназначенной для обработки нажатия пункта или подпункта меню с такими же задачами. Пусть, например, первая кнопка, уже расположенная на панели, дублирует нажатие подпункта *Ввод*

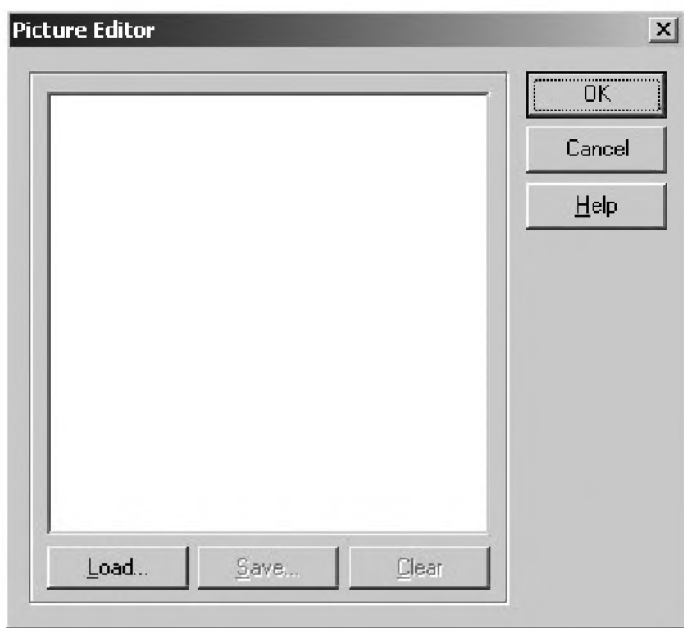


Рис. 9.12. Редактор картинок

N_Min пункта меню *Ввод пределов*. Для реализации этой цели в поле ввода события *OnClick* при помощи раскрывающегося списка выберете функцию *NMinClick*. Тело *этой же* функции можно вставить в заготовку функции, появляющейся при двойном щелчке по кнопке. Но последний способ увеличивает количество функций приложения, поэтому менее разумен.

Для подпункта *Ввод N_Max* (первого пункта меню) и второго пункта меню *Выход* создайте самостоятельно *свои* кнопки быстрого доступа. Все три кнопки украсьте картинками из указанного каталога. Не расстраивайтесь, если выбранные картинки плохо соответствуют назначению кнопок. Ведь эти операции выполнены *только* для тренировки. Удачно подобрать *готовую* иконку для кнопки часто весьма затруднительно. Поэтому обычно либо *модернизируют* наиболее подходящее изображение из набора картинок (или существующих стандартных кнопок каких-либо приложений), либо самостоятельно *создают* новую картинку. Оба последних варианта изготовления картинок осуществляются при помощи *Редактора Картинок*. Изложим поподробнее эти варианты.

Загрузите *любой* файл с *готовой* картинкой (из указанного выше каталога) в графический редактор *MS Paint*. В окне редактора увидите вид кнопки в *отжатом* и *нажатом* положении. Каждая картинка для кнопки располагается в квадратике со стороной в 16 пикселей, размеры суммарной картинки (вид кнопки в *двух* состояниях) поэтому составляют 16×32 пикселей. Указанные габаритные размеры изменять *нельзя*. Для первой кнопки разрабатываемого интерфейса сохраните копию файла с картинкой под новым именем *Min.bmp* (поскольку эта кнопка предназначена для ввода минимальной границы вводимого числа) в каталоге с исходным файлом картинки. Фон кнопки оставьте прежним, но вместо первоначальных изображений напишите при помощи попиксельного редактора (он включается при увеличенном масштабе) слово *Min* темно синим цветом для отжатой кнопки и жёлтым цветом для её нажатого положения. После окончательного сохранения загрузите созданное изображение на указанную кнопку прежним способом.

Аналогичные операции повторите для двух оставшихся кнопок. В результате у вас получится интерфейс, показанный на рис. 9.13.

Во втором способе разработки и модернизации картинок для кнопок *SpeedButton* применяется *Редактор Изображений Image Editor*. Этот редактор используется также при разработке изображений для кнопок *BitBtn*, а также курсоров и пиктограмм.

Вызов редактора осуществляется командой горизонтального меню: *Tools/Image Editor*. Окно редактора представлено на рис. 9.14.

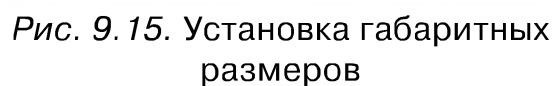
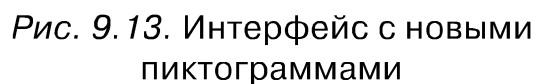
Этот графический редактор встроен в *Builder*, он позволяет сохранять созданные изображения не только в виде файлов, но и сразу включать их в файл ресурсов приложения. В этом заключается его отличие от других более мощных графических редакторов. Все его инструменты подобны растровому графическому редактору *Paint* и векторному графическому редактору в *Word*. Выполните команду в этом редакторе: *File/New/Bitmap File*. В окне выбора установите габаритные размеры разрабатываемой картинке: 16×32 пикселей (см. рис. 9.15).

По умолчанию предлагается 16 цветов для изготовления рисунка. После нажатия кнопки *ОК*, увидите изображение разрабатываемой кнопки в натуральную величину. С таким маленьким изображением работать очень неудобно, поэтому при помощи последовательного ввода команды *View/Zoom In* увеличьте изображение кнопки в *несколько* раз. После этого при помощи инструмента *карандаш* напишите слово, например, *Max* в обеих частях рисунка теми же цветами, что применялись для кнопки с надписью *Min*. В результате получится изображение, показанное на рис. 9.16. Сохраните его в файле с именем *Max.bmp* в упомянутом выше каталоге. Размещаются такие картинки на кнопках при помощи *Редактора Картинок Picture Editor*.

В заключение опишем ещё несколько полезных свойств кнопки *SpeedButton*. Эта кнопка может использоваться *не только* как обычная командная кнопка (типа *TButton* или *TBitBtn*), но и как кнопка с *фиксацией* нажатого состояния. Поэтому её ещё называют *Кнопкой с фиксацией*.

Если на интерфейсе имеется *несколько* кнопок, имеющих *одинаковое* (ненулевое) значение в свойстве *GroupIndex* (номер группы), то они образуют группу *взаимосвязанных* кнопок, из которых нажатой может быть *только одна*. При нажатии одной из кнопок группы она *фиксируется* в нажатом состоянии, а ранее нажатая кнопка переходит в отжатое состояние. Если значение свойства *AllowAllUp* (разрешено всем кнопкам принимать отжатое положение) равно *false*, то одна из кнопок группы *всегда* будет оставаться нажатой (последующие щелчки по ней указателем мыши *не переводят* её в отжатое положение). В случае, когда свойство *AllowAllUp* равно *true*, нажатая кнопка при щелчке по ней переходит в отжатое состояние, *все* кнопки группы одновременно могут быть в *отжатом* положении. Обращаем внимание на то, что выбор значения свойства *AllowAllUp* (*true* или *false*) для *одной* кнопки автоматически приводит к такому же значению этого свойства во *всех* остальных кнопках группы.

В ходе выполнения программы состояние отдельной кнопки определяется значением свойства *Down* (фиксировать в нажатом состоянии): если оно равно *true*, а



значение свойства *GroupIndex*>0, то кнопка остаётся в *нажатом* положении.

Если свойство *GroupIndex*= 0, то кнопка ведёт себя так же, как *Button* и *BitBtn*: при нажатии пользователем кнопки она погружается, а при опускании возвращается в нормальное состояние. В этом случае свойства *AllowAllUp* и *Down* не влияют на поведение кнопки.

Если *GroupIndex*>0, то кнопка при щелчке по ней погружается и остаётся в нажатом состоянии. При повторном щелчке по кнопке она *не переходит* в отжатое состояние (если это не осуществить программно). Если же *GroupIndex*>0 и *AllowAllUp*==*true*, то кнопка при щелчке по ней также погружается и остаётся в нажатом состоянии. Однако повторный щелчок переводит её в *отжатое* положение (этот режим работы применяется чаще).

Вопросы для самоконтроля

- ☐ В каких случаях используют файлы?
- ☐ Назовите два типа файлов *C++Builder*. Чем они отличаются?
- ☐ Как можно обработать заданные элементы файла последовательного доступа?
- ☐ Укажите способ получения доступа к любому байту файла с данными произвольного типа.
- ☐ Опишите три способа разработки картинок для кнопок быстрого доступа.
- ☐ Как назначить горячие клавиши подпунктам меню?

9.6. Задачи для программирования

Задача 9.1. Модернизируйте решение задачи восьмого урока таким образом, чтобы исходные данные находились в текстовом файле *Isx_f.txt*, а все результаты обработки вместе с исходными данными были записаны в текстовый файл *Rez_f.txt*. Повтор задачи восьмого урока: в матрице $M(4,3)$ найти сумму элементов третьей строки и сумму элементов второго столбца, отсортировать по возрастанию элементов упомянутые строку и столбец. Задачу решить с применением пользовательских функций.

Задача 9.2. Усовершенствуйте приложение *Матрица в двоичном файле* настоящего урока так (см. п. 9.4.3), чтобы показать на экране третий элемент третьей строки матрицы, прочитав его значение *непосредственно* из двоичного файла. Значение этого же элемента вывести на экран и из самой матрицы *iMatr[n][n]*. Назовите настоящее приложение *Задача 9.2*, а его форму *Zadacha_9_2*.

9.7. Варианты решений задач

Решение задачи 9.1.

Интерфейс приложения *Задача 9.1* показан на рис. 9.17, код файла реализации приводится ниже.

```

const int nn= 100;//Максимальное число строк и столбцов
typedef float fMas[nn];
typedef float fMatr[nn][nn];
fMas fSt_Mas, fCl_Mas;//Одномерные массивы
fMatr fM;//Матрица

TStringList *Spisok = new TStringList;
String Isx_Dan, Rez_Dan;

//Заносим данные из файла в матрицу и массивы
void Chtenie(
    fMatr & fMat, int iSt, int iCl, fMas & fMas_St, fMas & fMas_Cl,
    int nom_St, int nom_Cl)
{
    try
    {
        Spisok->LoadFromFile(Isx_Dan);
    }//try
    catch (...)
    {
        ShowMessage("Файл \" " + Isx_Dan + " \" не найден");
        Abort();
    }//catch
    //Объявляем переменные-указатели на тип char
    char *Yk_Char[nn], *Yk_Char_1[nn];
    //Записываем строки матрицы из файла Isx_Dan в массив строк Mas_St
    String Mas_Strok[nn];
    for (int ii= 0; ii< iSt; ++ii)
        Mas_Strok[ii]= Spisok->Strings[ii];
    Spisok->Clear();//Очистка переменной Spisok
    //Преобразуем переменную Mas_Strok[nn] типа
    //String в переменную-указатель на char массива Yk_Char[nn]
    for (int ii= 0; ii< iSt; ++ii)
    {
        Yk_Char[ii]= Mas_Strok[ii].c_str();
        //Каждую строку Yk_Char[ii] преобразуем в массив
        //чисел, усекая эту строку после каждого пробела
        String Stroka;
        Yk_Char_1[ii]= strtok(Yk_Char[ii], " ");
    }
}

```

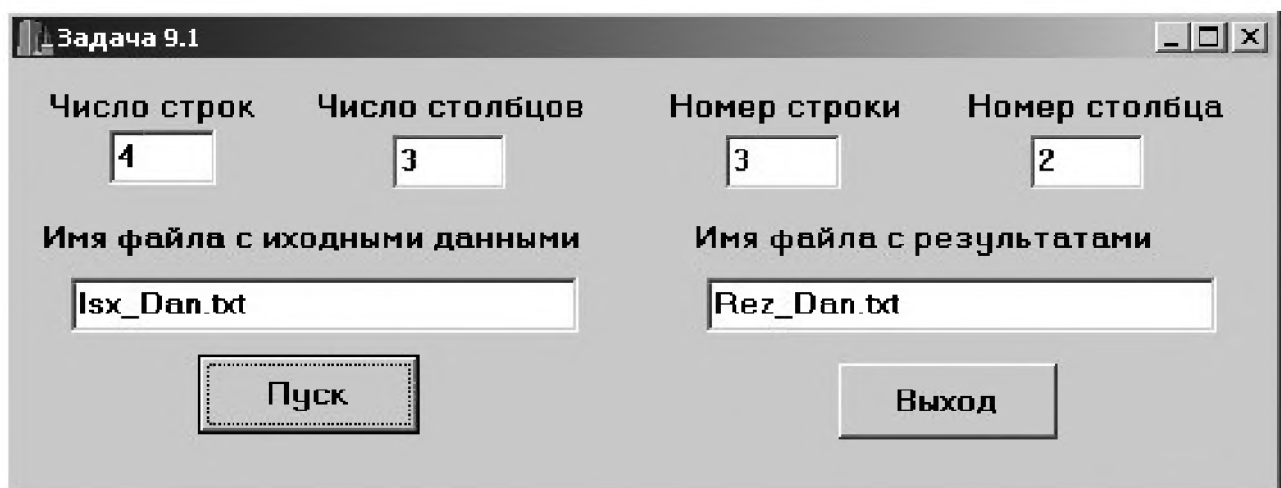


Рис. 9.17. Интерфейс приложения Задача 9.1

```

if (Yk_Char_1[ii])
{
    //Строку символов Yk_Char_1 преобразуем в строку String
    Stroka= String(Yk_Char_1[ii]);
    Spisok->Add(Stroka);
} //if
//Производим усечение оставшейся части строки
//Stroka по очередному пробелу
while (Yk_Char_1[ii])
{
    Yk_Char_1[ii]= strtok(NULL, " ");
    if (Yk_Char_1[ii])
    {
        Stroka= String(Yk_Char_1[ii]);
        Spisok->Add(Stroka);
    } //if
} //while
//Осуществляем запись данных в матрицу и массивы
for (int ij= 0; ij< iCl; ++ij)
{
    fMat[ii][ij]= StrToFloat(Spisok->Strings[ij]);
    //Переписываем строку
    fMas_St[ij]= fMat[nom_St - 1][ij];
    //Переписываем столбец
    fMas_Cl[ii]= fMat[ii][nom_Cl - 1];
} //for_ij
Spisok->Clear();
} //for_ii
} //Chtenie

```

```

void Sort_Mas(fMas & fMa, int iDl)
//Осуществляем линейную сортировку
{
    for (int ij= 0; ij< iDl - 1; ++ij)
        for (int iCan= ij+1; iCan< iDl; ++iCan)
            if (fMa[ij]> fMa[iCan])
            {
                float fByf= fMa[ij];
                fMa[ij]= fMa[iCan];
                fMa[iCan]= fByf;
            } //if
} // Sort_Mas

```

```

float Sym_Mas(float fMa[], int iDl)
{
    //Осуществляем суммирование массива fMass
    float fS= 0;
    for (int ij= 0; ij< iDl; ++ij)
        fS += fMa[ij];
    return fS; //Возврат результата в точку вызова
} //Sym_Mas

```

```

void Vuvod_Matr(fMatr fMat, int iSt, int iCl)
{

```

```

    for (int ii= 0; ii< iSt; ++ii)
    {
        String Stroka= ""; //Очистка не обязательна
        for (int ij= 0; ij< iCl; ++ij)
            Stroka += FloatToStrF(fMat[ii][ij],
                                   ffFixed, 10, 1) + " ";

        Spisok->Add(Stroka);
    } //for_ij
} // Vuvod_Matr

void Vuvod_Mas(fMas fMa, int iDl)
{
    String Stroka= ""; //Очистка не обязательна
    for (int ij= 0; ij< iDl; ++ij)
        Stroka += FloatToStrF(fMa[ij], ffFixed, 10, 1) + " ";
    Spisok->Add(Stroka);
} // Vuvod_Mas

void __fastcall TZadacha_9_1::PyskClick(TObject *Sender)
{
    const int in_Str= StrToInt(n_Str->Text),
              in_Cl= StrToInt(n_Cl->Text),
              // Выбор номеров строки и столбца
              inom_Str= StrToInt(nom_Str->Text),
              inom_Cl= StrToInt(nom_Cl->Text);
    Isx_Dan= Isxodn_Fajl->Text; //Ввод имён файлов
    Rez_Dan= Rezylt_Fajl->Text;

    Chtenie(fM, in_Str, in_Cl, fSt_Mas, fCl_Mas, inom_Str, inom_Cl);
    Spisok->Add("Исходная матрица");
    Vuvod_Matr(fM, in_Str, in_Cl);

    Sort_Mas(fSt_Mas, in_Cl);
    Sort_Mas(fCl_Mas, in_Str);
    Spisok->Add("Строка " + IntToStr(inom_Str) + " после сортировки");
    Vuvod_Mas(fSt_Mas, in_Cl);

    Spisok->Add("Столбец " + IntToStr(inom_Cl) + " после сортировки");
    Vuvod_Mas(fCl_Mas, in_Str);

    float fS= Sym_Mas(fSt_Mas, in_Cl);
    Spisok->Add("Сумма элементов " + IntToStr(inom_Str) +
               " строки равна " + FloatToStrF(fS, ffFixed, 10, 2));

    fS= Sym_Mas(fCl_Mas, in_Str);
    Spisok->Add("Сумма элементов " + IntToStr(inom_Cl) +
               " столбца равна " + FloatToStrF(fS, ffFixed, 10, 2));
    //Создаём файл Rez_Dan и записываем в него значения из объекта Spisok
    Spisok->SaveToFile(Rez_Dan);
    ShowMessage("Файл с итоговыми результатами готов!");
} //PyskClick

void __fastcall TZadacha_9_1::VuxodClick(TObject *Sender)
{

```

```
delete Spisok;  
Close();  
} //VuxodClick
```

Решение задачи 9.2.

Тело функции *PyskClick* для приложения **Задача 9.2**:

```
const int n= 5;  
randomize();  
int iMatr[n][n];  
for (int ii= 0; ii<n; ++ii)  
    for (int ij= 0; ij<n; ++ij)  
        iMatr[ii][ij]= random(n*n+1);  
int iNomer_Dv_F;  
String F_Dvoich= "Dvoichn_f",  
        F_Tekst = "Tekstov_f";  
iNomer_Dv_F= FileCreate(F_Dvoich);  
if (iNomer_Dv_F== -1)  
{  
    ShowMessage("Файл не удалось создать");  
    Abort();  
} //if  
FileWrite(iNomer_Dv_F, &iMatr, sizeof(iMatr));  
int Chislo; //Для хранения элемента матрицы  
//Устанавливаем указатель файла перед требуемым  
//элементом матрицы (5 + 5 + 2= 12).  
FileSeek(iNomer_Dv_F, (int)sizeof(int)*12, 0);  
//Считываем из файла в переменную Chislo 4 байта данных.  
FileRead(iNomer_Dv_F, &Chislo, sizeof(Chislo));  
ShowMessage("Chislo= " + IntToStr(Chislo)+  
            "    iMatr[2][2]= " + IntToStr(iMatr[2][2]));  
FileClose(iNomer_Dv_F);
```




Урок 10. Указатели и динамические переменные

10.1. Вводные замечания и основные определения

На предшествующих уроках неоднократно использовались указатели и динамические переменные. Наконец, пришла пора подробно разобраться с этим типом данных. Прежде всего, введём понятия статическая и динамическая память.

Статическая память это блок оперативной памяти, который *Builder* выделяет *на всё время* работы приложения для размещения кода программы и её переменных. Переменные, размещённые в такой памяти, называются статическими. Непрерывная область ячеек статической памяти именуется *сегментом данных*.

Если число переменных и их объём в приложении трудно предсказуем, то приходится оперативную память для них выделять с запасом. Ясно, что при этом в одних случаях запрошенная статическая память будет использоваться не полностью, а в других – скажется её недостаток и произойдёт остановка работы программы.

Указанные проблемы существенно уменьшаются с применением динамических переменных. *Динамические переменные* (в общем случае объекты) обычно порождаются в программе лишь *непосредственно перед* их применением. Когда необходимость в существовании таких переменных отпадает, оперативную память, занятую под их размещение, можно освободить. В связи с возможностью оперативного порождения и уничтожения, рассматриваемые переменные называются динамическими, а область памяти, в которой они расположены, именуется *динамической памятью* или кучей (*heap*).

Любую статическую или динамическую переменную можно представить в виде двух частей, двух блоков ячеек, которые, как правило, не находятся в смежных областях памяти. Первый блок ячеек, объёмом в 4 байта, ассоциируется с *именем* переменной, он как для статических, так и динамических переменных всегда расположен в сегменте данных. В этом блоке также хранится информация о *типе* переменной. Второй блок используется для размещения *значения* самой переменной, его объём определяется типом объявленной переменной. Этот *непрерывный* блок ячеек будем называть *объектом переменной* (или её телом).

Каждая ячейка оперативной памяти компьютера представляет собой физическое устройство, в котором можно записать лишь байт информации. Все такие ячейки имеют определённый адрес – местоположение в оперативной памяти

(среди других ячеек). Адрес *начальной ячейки* объекта переменной записывается (хранится) в первом блоке ячеек, связанном с её именем и типом. Этот адрес служит адресом и для всего объекта переменной.

Как уже не раз отмечалось, количество ячеек памяти для хранения объекта заданной переменной определяется её типом (хранимым в адресной части переменной). Адрес *любой* ячейки (или байта) объекта всегда можно определить, поскольку ячейки объекта представляют собой *непрерывный* блок, а адрес начальной ячейки хранится в адресной части переменной. Упомянутые блоки переменной представим в виде прямоугольников, соединённых стрелкой, как показано на рис. 10.1.

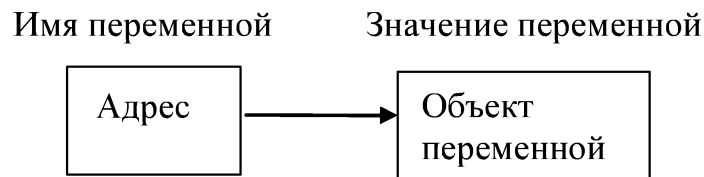


Рис. 10.1. Представление произвольной переменной

Стрелку назовём *указателем* переменной, она указывает на объект переменной. Для статической переменной этот объект расположен в статической памяти (в сегменте данных), а для динамической переменной – в куче. Ранее доступ к объекту переменной (при записи или чтении её значения) осуществлялся *только* при помощи имени переменной, поскольку в статической переменной под именем подразумевается содержимое её объекта. Однако адрес объекта такой переменной *всегда* можно узнать при помощи упоминаемой ранее операции взятия адреса «&».

Для хранения адреса объекта переменной предназначен специальный тип данных, называемый *указателем*. В переменных такого типа могут храниться *только* адреса ячеек памяти. При попытке записать туда данные других типов *Builder* сообщит, что он не в силах преобразовать тип конкретной переменной к требуемому типу указателя. Такое преобразование не может осуществиться *автоматически*, однако оно *выполнимо* при помощи *явного преобразования типа*, о чём уже упоминалось ранее и более подробно рассказывается ниже. Сейчас же ещё раз покажем, как объявляется указатель *Yk*, например, на тип *int* (аналогичным образом можно объявить указатель на *любой* тип данных):

```
int *Yk;
```

Теперь в переменную *Yk* можно записать адрес *любой* переменной, типа *int*:

```
int ix = 54; //Объявление и инициализация переменной
Yk = &ix; //Запись адреса переменной ix в указатель
```

В целях компактности кода программы процесс объявления указателя и его инициализацию можно совместить:

```
int *Yk = &ix;
```

Инициализацию любого указателя допустимо осуществить ещё и так:

```
int *Yk(&ix);
```

Далее в основном будем применять первый вариант инициализации, поскольку из двух *полностью эквивалентных* вариантов он представляется нам более ясным. После инициализации указателя *Yk* (любым из приведенных выше способов) доступ к объекту переменной *ix* или объекту указателя *Yk* (физически это один и тот же объект, блок ячеек памяти) можно осуществить через указатель или переменную:

```
ShowMessage(*Yk); //54  
ShowMessage(ix); //54
```

В обоих случаях на экран выводится число 54. Для *обращения* к объекту указателя *Yk* в первой функции *ShowMessage* использована операция *разыменования* (или *разадресации, обращения по адресу, косвенной адресации*) – поставлена звёздочка перед именем указателя.

В объявленный указатель *Yk* можно записать адрес *не только статического, но и динамического* объекта. Последняя операция применяется значительно чаще. В этом случае в объекте будут находиться неприсвоенные значения (предшествующие значения ячеек, интерпретируемые по выбранному типу указателя). Перейдём теперь к процессу *порождения* динамической переменной.

Создать динамическую переменную возможно и различными устаревшими способами, однако ниже они упоминаться не будут. Для *выделения* памяти под объект динамической переменной будем использовать *только* операцию *new*. Вначале в указатель *Yk* запишем адрес начальной ячейки блока ячеек из 4 байт, расположенных в куче. В этот блок вначале ничего не записываем – т.е. не инициализируем переменную. Затем в упомянутый объект динамической переменной запишем число 58:

```
int *Yk= new int;  
*Yk= 58;  
ShowMessage(*Yk); //58
```

Теперь на экране появится число 58. Первую строку предшествующего кода допустимо записать в виде двух строк:

```
int * Yk;  
Yk= new int;
```

Создание динамической переменной разрешается совмещать с её инициализацией:

```
int *Yk= new int(8);
```

Операция инициализации имеет также и несколько другой синтаксис:

```
int *Yk= new (int) (8);
```

В любом из вариантов синтаксиса на экране появится число 8 при вызове функции:

```
ShowMessage(*Yk); //8
```

Впереди круглых скобок с иницилирующим значением в обоих вариантах синтаксиса пробел может отсутствовать, стоять один или более пробелов.

Очередной пункт хорошего стиля программирования состоит в *уничтожении* динамических переменных в конце программы, в конце функции, в которой они порождались, или сразу же, когда в их существовании отпала необходимость. При этом ресурсы памяти, занятые динамическими переменными, освобождаются. Если не уничтожать динамические переменные после их использования (обработки), то в некоторых программах может возникнуть ситуация, когда для размещения очередной динамической переменной или объекта нет свободной памяти.

По завершению приложения занятые им ресурсы памяти *автоматически* корректно освобождаются, однако программист должен выработать привычку *самостоятельно* уничтожать порождённые им динамические переменные. Для *уничтожения* динамических переменных применяется операция *delete*, вот пример её использования:

```
delete Yk;
```

Эта операция *освобождает* память, разрывает первый (адресный) блок динамической переменной со вторым, где хранится её объект. После завершения этой операции в адресном блоке оказывается неприсвоенное значение, стрелка этого блока ни на что не указывает. Поэтому настоятельно рекомендуется записать в указатель 0 или символьную константу *NULL*. Это адрес *несуществующей* ячейки, и по нему программно (при помощи оператора *if*) можно решать вопрос, направлен ли указатель на какую либо ячейку или нет. Если же попытаться в программе использовать неинициализированный указатель, то последствия могут быть *непредсказуемыми*. Вот так рекомендуется корректно выполнять операцию уничтожения динамической переменной:

```
delete Yk;
```

```
Yk= NULL;
```

```
// или Yk= 0;
```

Более наглядным является использование текстовой константы *NULL*.

Адрес объекта динамической переменной не обязательно размещать в указателе. Его можно записать и в обычную переменную вот таким способом:

```
int ix = *new int (16);
```

```
ShowMessage(ix); //16
```

Как и ранее инициализацию динамического объекта переменной *ix* можно выполнить и позже, не в момент (не в строке) его порождения, служебное слово *int* (тип переменной), расположенное за операцией *new*, также допускается помещать в круглые скобки. Далее переменную *ix* используют, как и обычную переменную. Правда на самом деле эта переменная стала не совсем обычной, ведь её объект теперь находится в куче и разорвать его связь с *ix* операцией *delete* нельзя. От такого варианта размещения объекта динамической переменной рекомендуем воздерживаться в своей практике.

10.2. Иллюстрация применения указателей

Наш опыт преподавания языков программирования показывает, что учащиеся далеко не сразу глубоко осознают назначение указателей и правила работы ними. Поэтому ниже работу с указателями проиллюстрируем простыми примерами с подробными комментариями и наглядными рисунками.

```
int *Yk_ix, *Yk_iy; //Объявлены указатели на тип int
```

Этим кодом сообщается, чтобы *Builder* выделил два блока ячеек с именами соответственно *Yk_ix* и *Yk_iy* для возможности размещения в них адресов двух объектов переменных типа *int*. Таким образом, в этих ячейках будут храниться не только упомянутые выше адреса объектов, но и информация о том, какой тип (и объём) у объектов этих переменных. В приведенные выше указатели можно записать адреса как статических, так и динамических объектов.

```
//Выделение динамических объектов и запись
//их адресов в указатели
Yk_ix= new int;
Yk_iy= new int;
```

В куче выделяются два блока ячеек динамической памяти объёмом по 4 байт каждый, адреса начальных ячеек этих блоков записываются в соответствующие указатели, в порождённых объектах пока находятся неприсвоенные значения (см. рис. 10.2).



Рис. 10.2. Объявили указатели и связали их с пустыми объектами

```
//Записываем значения в динамические объекты
*Yk_ix= 27;
*Yk_iy= 54;
ShowMessage(*Yk_ix); //27
ShowMessage(*Yk_iy); //54
```

На экране последовательно появятся числа 27 и 54. Изменения в динамических переменных иллюстрирует рис. 10.3.



Рис. 10.3. Наполнили динамические объекты конкретными значениями

```
//Записываем адрес из указателя Yk_iy в указатель Yk_ix
Yk_ix= Yk_iy;
ShowMessage(*Yk_ix); //54
ShowMessage(*Yk_iy); //54
```

Теперь в указателях *Yk_ix* и *Yk_iy* хранится адрес одной и той же ячейки динамической памяти (см. рис. 10.4). На экран дважды будет выведено число 54. Динамический объект, содержащий число 27, теперь стал *недоступен* для программиста, память под его размещение освобождается *только* после завершения работы приложения.

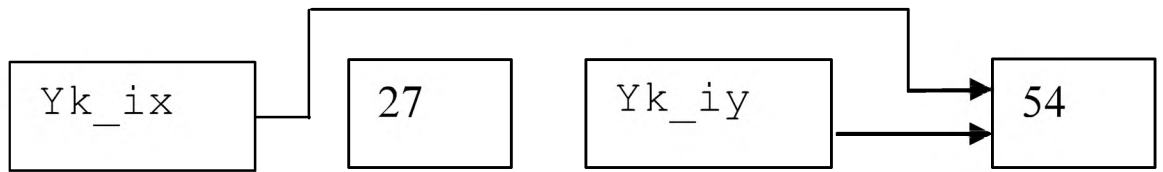


Рис. 10.4. Переключение указателя с одного объекта на другой

```
//Корректный разрыв связи указателя с динамическим объектом
Yk_iy= NULL;
```

Итоговые связи ячеек памяти иллюстрирует рис. 10.5.

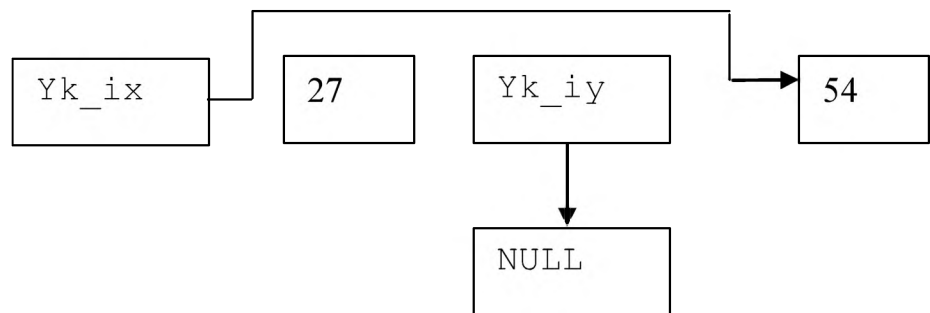


Рис. 10.5. Корректный разрыв связи указателя с динамическим объектом

При помощи указателей в динамической памяти можно разместить объект переменных *любой* типов как основных, так и составных.

10.3. Указатели на различные типы данных

10.3.1. Виды указателей

Указатели делятся на три вида – указатели на объекты разных типов данных (кроме типа *void*), указатели на тип *void* и указатели на функции. Указатель на функцию – это адрес ячейки в сегменте кода оперативной памяти, с которой начинается исполняемый код функции. Присвоение указателям на тип переменных указателей на функции (и наоборот) *недопустимо*. Для указателей на функции не разрешены какие-либо арифметические операции. В то же время указатели на объекты переменных могут вычитаться друг из друга, их разрешается складывать с константами, из них допустимо вычитать константы, к ним применимы операции декримента и инкримента, операции сравнения. Эти и другие операции с указателями на объекты рассматриваются в следующем разделе. Об указателях на функции подробнее расскажем в разделе 10.8, сейчас же займёмся *разновидностями*

стями указателей на объекты и соотношениями между указателями на различные типы данных.

10.3.2. Операции с однотипными и разнотипными указателями

Как уже отмечалось, объём адресной части указателя *не зависит* от объёма объекта на который он указывает. Этот факт наглядно подтверждает приведенный ниже код.

```
int ix= 54;
double dx= 54.4861;
int *Yk_int= &ix;
double *Yk_double= &dx;
ShowMessage(sizeof(Yk_int)); //4
ShowMessage(sizeof(Yk_double)); //4
ShowMessage(sizeof(*Yk_int)); //4
ShowMessage(sizeof(*Yk_double)); //8
```

В результате её работы на экране появится вначале три раза число 4, а затем число 8. Первые две четвёрки – это объёмы в байтах адресных частей введенных указателей, последующая четвёрка и восьмёрка являются объёмами объектов, предназначенных для хранения соответственно чисел типа *int* и *double*.

Однотипным указателям *можно* присваивать значения друг друга. При записи в адресную часть указателя *нового адреса* (из однотипного указателя) происходит *разрыв* связи с его прежним объектом и устанавливается связь с другим объектом, адрес которого скопирован в адресную часть принимающего указателя. При таком присваивании оба указателя указывают *на один и тот же блок* памяти. В примере, показанном ниже, на экран будет выведено число 27.

```
int ix= 54, iy= 27;
int *Yk_ix= &ix, *Yk_iy= &iy;
Yk_ix= Yk_iy;
ShowMessage(*Yk_ix); //27
```

Все типы указателей, кроме типа *void*, могут быть связаны с объектами соответствующего типа, как отмечалось, информация о размере объекта переменной хранится в адресной части указателя *совместно* с адресом начальной ячейки его объекта. Однако у указателя на тип *void* объект *отсутствует*. В указатель на тип *void* *разрешается* записывать адреса данных *произвольного* типа и указатели на *произвольный* тип, *кроме* указателей на функции. Указатель на тип *void* можно (при определённых условиях) присваивать указателям на другие типы данных.

Присваивание указателям на тип *void* указателей произвольных типов оказывается возможным в связи с тем, что операция преобразования типов в ходе присваивания значений в такой указатель, выполняется *по умолчанию*. Фактически такое преобразование заключается лишь в том, что в указатель на тип *void* записывается *только* адрес объекта назначаемого указателя, а информация об объёме и типе объекта этого указателя *игнорируется*. Именно по этому при присваива-

нии указателя на *void* указателю на другой тип данных *требуется* явное преобразование типа:

```
int ix= 54;
void * Yk_void = &ix;
int *Yk_int = (int *)Yk_void;
ShowMessage(*Yk_int);//54
```

На экране покажется число 54. При этом принимающий указатель должен быть однотипен с исходным объектом. Если же при явном преобразовании применить тип, отличный от типа объекта, адрес которого ранее был записан в указатель на тип *void* (в нашем примере это тип *int*), то получите *неприсвоенное* значение.

```
float *Yk_float = (float *)Yk_void;
ShowMessage(*Yk_float);//Неприсвоенное значение
```

Присваивать друг другу значения *разнотипных* указателей (за исключением типа *void*) можно *только* с использованием *явного преобразования* типа. При этом в принимающий указатель записывается адресная часть присваиваемого указателя, а информация о размере объекта присваиваемого указателя операцией преобразования типа заменяется на размер типа принимающего указателя. Значение же присвоенного объекта оказывается как правило неопределённым (неприсвоенным). Поэтому *не рекомендуется* злоупотреблять явным преобразованием типов, если же оно всё же использовано, то необходимо очень осторожно относиться к получаемым результатам. Сказанное поясняет следующий фрагмент программы:

```
double dx= 54.5454;
double *Yk_double= &dx;
int *Yk_int = NULL;//Хороший стиль программирования —
//инициализация всех указателей в ходе их объявления
Yk_int = (int *)Yk_double;
ShowMessage(*Yk_int);//неприсвоенное значение
```

В указателе *Yk_int* записана адресная часть указателя *Yk_double*, в результате старый адрес замещается новым, однако на экране появится неприсвоенное значение объекта указателя *Yk_int*. Хорошим стилем программирования является инициализация *всех* указателей в ходе их объявления: это исключает использование в приложении указателей с неприсвоенными значениями. Поэтому указателю *Yk_int* присваивается при его объявлении значение *NULL*, которое в следующей строке программы заменяется адресом конкретной ячейки памяти.

10.3.3. Указатель на указатель

Как уже упоминалось, адресная часть всякого указателя состоит из четырёх ячеек памяти, из четырёх байт. Начальная ячейка этого блока также имеет свой адрес (он является и адресом этого указателя), который определяется операцией взятия адреса. Поэтому можно объявить указатель на указатель. Объектом такого указателя служит адресная ячейка первого указателя. На такой указатель можно объявить ещё один указатель и так далее. Ограничений этому процессу нет. Ниже приведен фрагмент программы, иллюстрирующий этот процесс из четырёх

последовательных этапов. Для получения значения первоначального объекта следует впереди идентификатора последнего указателя (полученного на последнем этапе) поставить столько звёздочек, сколько этапов было в описанном процессе.

```
int ix= 54;
int *Yk_int= &ix;
int **Yk_Yk_1= &Yk_int;
int ***Yk_Yk_2= &Yk_Yk_1;
int ****Yk_Yk_3= &Yk_Yk_2;
ShowMessage(****Yk_Yk_3); //54
```

На экране появится число 54. Удобно в операции разыменования использовать круглые скобки, которые помогают разобраться в объектах указателей различных этапов: *ShowMessage(*((* (*Yk_Yk_3))*)). Не рекомендуем такие головоломки применять в программах без острой необходимости.

10.3.4. Константный указатель на константные данные

При определении такого указателя как адрес объекта, на который он указывает, так и значение самого объекта должны быть неизменными. Для достижения этих целей дважды используется модификатор *const*. В нижеприведенном коде программы значение константы *M_PI* может быть выведено на экран. Однако удаление знаков однострочного комментария (закомментированная часть кода) приведёт к ошибке: *Cannot modify a const object*. Это свидетельствует о том, что константный указатель на константные данные доступен *только для чтения, ни одну* из его составляющих изменять *нельзя*.

```
double Pi= M_PI;
double const *const Const_Yk_Const_Pi= &Pi;
ShowMessage(*Const_Yk_Const_Pi); //3.1415
double dx= 54.5454;
//Const_Yk_Const_Pi= &dx; //Cannot modify a const object
//*Const_Yk_Const_Pi= dx; //Cannot modify a const object
```

10.3.5. Неконстантный указатель на константные данные

При программировании применяется неконстантный указатель на константные данные. В этом случае разрешается изменять *только* адресную часть указателя, а содержимое его объекта должно оставаться неизменным. Приведём поясняющий код.

```
int ix_1= 1, ix_2= 2;
int const *Yk_Const_int= &ix_1;
Yk_Const_int= &ix_2;
ShowMessage(*Yk_Const_int); //2
*Yk_Const_int= ix_1; //Cannot modify a const object
```

На экране выводится значение 2, а затем появится сообщение об ошибке.

10.3.6. Константный указатель на неконстантные данные

В константном указателе на неконстантные данные можно изменять содержимое объекта, на который указывает указатель, но *адрес* этого объекта *неизменен* (содержимое адресной части изменять нельзя). Как отмечалось ранее (см. восьмой урок), по умолчанию любой массив является таким указателем.

```
int ix_1= 1, ix_2= 2;
int *const Const_Yk_int= &ix_1;
*Const_Yk_int= ix_2;
ShowMessage(*Const_Yk_int); //2
Const_Yk_int)= &ix_2; //Cannot modify a const object
```

На экран будет выведено число 2, а затем сообщение об ошибке: *Cannot modify a const object*.

10.4. Адресная арифметика с массивами

Как не раз отмечалось, массивы по умолчанию являются константными *указателями* на неконстантные данные. Поскольку любой массив поэтому (прежде всего) является *указателем*, поэтому в идентификаторе (имени) массива хранится *адрес массива* или адрес его начальной ячейки. Пусть задан массив *Mas*:

```
const int n= 5;
int Mas[n]= {27, 5, 2004, 14, 3};
Номера ячеек: 0, 1, 2, 3, 4.
```

Для массива *Mas* справедливы соотношения:

```
Mas== &Mas== &Mas[0];
```

Однако массив – это не просто указатель, а *константный указатель*, из чего следует, что *адрес* его объекта изменить *нельзя*. Приведём пример для прежнего массива *Mas*:

```
int ij= 0;
Mas= &ij; //Ошибка
```

Для такого кода *Builder* выведет следующее сообщение о допущенной ошибке: *Lvalue required*. Оно означает, что имя массива (без индексов в квадратных скобках) может стоять *только справа* от оператора присваивания.

Приведём фрагмент кода программы для массива *Mas*, введенного и инициализированного выше:

```
ShowMessage(Mas[2]); //2004
ShowMessage(*(Mas + 2)); //2004
ShowMessage(2[Mas]); //2004
```

Значения, последовательно появляющиеся на экране, приведены в качестве комментария после функции *ShowMessage*. Первое сообщение достаточно очевидно, поскольку выводится на экран третий элемент массива, который в C++ имеет номер 2.

Поскольку имя массива – это адрес его нулевой ячейки, то сумма ($Mas + 2$) трактуется как адрес компонента массива Mas с индексом 2. Дело в том, что в этой сумме под числом 2 понимается смещение от адреса нулевой ячейки массива на количество байт, равных произведению $2 \cdot \text{sizeof}(Mas[0])$, где $\text{sizeof}(Mas[0])$ – объём компонента массива (в нашем случае $\text{sizeof}(int) == 4$). Таким образом, разадресация $*(Mas + 2)$ является третьим элементом массива.

В общем случае действие операции $Mas[indeks]$ происходит так $*(Mas + indeks)$. При этом *фактическое* значение указателя Mas изменяется на величину $indeks \cdot \text{sizeof}(int)$. Если указатель на определённый тип данных увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженной на размер объекта данного типа.

Вычитать и складывать константы с указателями разумно *только* в случае массивов и указателей на массивы (см. следующий раздел). Даже при *последовательном* порождении *однотипных* переменных их объекты, как правило, размещаются в памяти *не последовательно* друг за другом, поэтому при добавлении и вычитании констант далеко не всегда можно указатель переместить на нужную переменную (на объект требуемой переменной).

Поскольку результат сложения не зависит от перемены мест слагаемых, то: $Mas[indeks] == *(indeks + Mas) == indeks[Mas]$. Именно поэтому третье сообщение в приведенном выше примере также было 2004. Упомянутое свойство массивов не рекомендуем применять на практике, поскольку его использование делает программы очень уж неясными.

Массивы это не совсем обычные указатели, например, для прежнего массива Mas операция $\text{sizeof}(Mas)$ возвращает не объём адресной части указателя (4 байта), а *размер всего массива* в байтах (размер объекта указателя): $\text{sizeof}(Mas) == \text{sizeof}(Massiv_int[0]) \cdot n == 4 \cdot 5 = 20$ байт. Таким образом, в операции sizeof массив «забывает», что он является указателем, и это позволяет легко вычислить объём его объекта или объём всего массива.

Уважаемые читатели, то с чем вы познакомитесь ниже, настоятельно *не рекомендуем* применять в программах, чтобы не превращать их в головоломки. Однако освоение нижеследующего материала весьма полезно для *полной ясности* в понимании устройства массивов, как константных указателей. Рассмотрим ещё две строки кода для прежнего массива Mas .

```
ij= 0;  
ShowMessage(*(Mas + ++ij)); //5, элемент Mas с номером 1
```

После операции $++ij$ (префиксная форма инкремента) в ij находится единица ($0 + 1 == 1$), поэтому указатель Mas передвинут с нулевого элемента на первый элемент, равный 5 (разадресация элемента с первым индексом).

```
ShowMessage(*(Mas + --ij)); //27, 1-й элемент Mas
```

Операция $--ij$ (префиксная форма декремента) уменьшила значение ij от единицы до нуля, что привело к появлению на экране элемента массива с индексом нуль, равного 27.

Приведём очередные три строки кода.

```
ij= 2;
ShowMessage( ++ij[Mas]); //2005
ShowMessage( --ij[Mas]); //2004
```

Для выяснения результатов работы префиксной формы инкремента и декремента выполним вначале преобразования: $ij[Mas] = Mas[ij]$. Поэтому в ходе операции $++Mas[ij]$ вначале выбирается элемент массива 2004 (с индексом 2), а затем к нему добавлена единица ($2004 + 1 == 2005$). Но значение 2005 *не только* выводится на экран, оно ещё и *записывается* в элемент массива с индексом 2. Далее префиксный декремент $--j[Mas]$ уменьшает новое значение (2005) элемента с индексом 2 массива *Mas* на единицу, после чего изменённое значение (2004) записывается в массив и выводится на экран.

Ещё три строки кода:

```
ij= 2;
ShowMessage(ij++[Mas]); // 2004
ShowMessage(Mas[ij]); //14
```

Постфиксная форма инкремента и декремента изменяет *не текущее, а последующее* значение индексов выводимых компонент массива. Поэтому первый вызов функции *ShowMessage* выводит значение (2004) элемента массива *Mas* с индексом 2, после чего постфиксная форма инкремента увеличивает *индекс* массива на единицу ($2 + 1 == 3$), а второй вызов функции *ShowMessage* выводит элемент массива *Mas* с индексом 3.

Наконец, последние три строки:

```
ij= 2;
ShowMessage(ij--[Mas]); //2004
ShowMessage(Mas[ij]); //5
```

Во второй строке выводится *текущее* значение элемента массива *Mas* с индексом 2 и уменьшается его индекс на единицу ($2 - 1 == 1$), поэтому в третьей строке выводится элемент массива с индексом 1, равный 5.

10.5. Адресная арифметика с указателями на массивы

Настроим указатель *Yk_Mas* на прежний массив *Mas* (см. раздел 10.4), после чего определим объём адресной части и объекта этого указателя.

```
int *Yk_Mas; //Объявлен указатель на тип int
Yk_Mas= Mas; //Указатель Yk_Mas настроен на массив Mas
ShowMessage(sizeof(Yk_Mas)); //4
ShowMessage(sizeof(*Yk_Mas)); //4
```

На экране последовательно будут выведены две четвёрки. Первая четвёрка – это размер адресного блока ячеек указателя *Yk_Mas*, а вторая – объём начальной ячейки массива *Mas*, на который был настроен указатель *Yk_Mas*. Таким образом, в отличие от массива, указатель на массив является *обычным указателем*, операция *sizeof*, применённая для его имени, возвращает не объём массива, как это

было в случае с массивом, а *объём адресной* части самого указателя на массив. Объём же разадресованного объекта указателя на массив (**Yk_Mas*), вычисляемый операцией *sizeof*, также как и в случае массива, возвращает размер нулевой ячейки (фактически объём типа массива).

Операция *&Yk_Mas* возвращает адрес указателя *Yk_Mas*, а не адрес начала массива *Mas*, на который настроен указатель. В подтверждение этого утверждения нижеследующий фрагмент программы выведет сообщение: «Это разные адреса».

```
if (&Yk_Mas != &Mas)
    ShowMessage("Это разные адреса");
```

Доступ к любому элементу массива *Mas* можно осуществить *ещё двумя* способами. Первый из них реализуется при помощи смещения указателя и его разадресации:

```
ShowMessage(*(Yk_Mas + 2)); //2004
```

Во втором способе используются квадратные скобки после имени указателя на массив: указатели на массивы можно индексировать точно так же как и массивы:

```
ShowMessage(Yk_Mas[2]); //2004
```

На экран дважды выводится число 2004. Как видите, применение операции *квадратные скобки* к указателям на массивы вполне *допустимо*, однако при этом ясность программы несколько ухудшается. Идентификатор указателя на массив *Yk_Mas* теперь является *синонимом* (псевдонимом, другим именем) имени самого массива. Это позволяет доступ к компонентам объекта массива *Mas* (приведенная формулировка при первом чтении звучит довольно странно) осуществить при помощи имени массива (*Mas*) и имени указателя на массив (*Yk_Mas*). Не следует забывать, что во всех способах доступ осуществляется *к одному и тому же* непрерывному блоку ячеек оперативной памяти компьютера.

Можно объявить не один, а произвольное число указателей и каждый из них направить на один и тот же массив (например, на начало массива *Mas*). В этом случае порождается целый ряд указателей, направленных на один и тот же массив (на один и тот же объект массива), а не ряд массивов с разными именами, но с одинаковыми значениями компонент.

Для указателей на массив весьма полезно ещё одно действие адресной арифметики: *вычитание указателей*. Ниже определим два указателя на тип *int* (*Yk_1* и *Yk_2*) и настроим их на *различные* компоненты массива *Mas*.

```
int *Yk_1, *Yk_2;
int ix;
Yk_1 = Mas + 2; //Настроен на компоненту с индексом 2
Yk_2 = Mas + 4; //Настроен на компоненту с индексом 4
ix = Yk_2 - Yk_1;
ShowMessage(ix); //2
```

В результате на экране покажется число 2 – количество *компонент* массива *Mas* между компонентами с индексами 2 и 4. Далее перестроим указатели *Yk_1* и *Yk_2*:

```
Yk_1= Mas + 0; //Настроен на начало массива Mas
```

```
Yk_2= Mas + 5; //Адрес конца массива Mas
```

В окне вывода функции *ShowMessage(ix)* теперь появится число 5 – число элементов массива между адресом его ячейки с индексом 0 и адресом ячейки, расположенной за *последней* ячейкой с индексом 4 (напоминаем, что в использованном массиве всего 5 ячеек). Разность адресов *ix* измеряется в единицах, равных размеру типа массива. Поэтому для вычисления объёма массива, выраженного в байтах, следует полученную разность умножить на объём типа. В нашем примере объём массива равен 20 байтам ($5 \cdot 4 = 20$).

Как отмечалось выше, указатели на массивы можно складывать с положительными константами. При этом указатель перемещается на число компонент своего типа в положительном направлении. При вычитании из указателя положительных констант (или сложении отрицательных), указатель смещается в обратном направлении, номер компоненты массива при этом уменьшается, приближается к нулевой ячейке. Если разность указателей, настроенных на разные ячейки массива, равна отрицательному числу, это означает, что «вычитаемый адрес» находится далее от адреса нулевой ячейки массива, чем адрес, из которого производится вычитание. Сложение указателей не имеет смысла и запрещено синтаксисом языка программирования. В случае нарушения этого запрета появляется сообщение: *Invalid pointer addition* (недопустимая операция сложения с указателем).

Напоминаем, что операция алгебраического сложения констант с указателями и вычитание указателей имеет смысл *только* в случае, когда они направлены на компоненты *одного и того же* массива. Массивы, даже порождаемые последовательно, могут располагаться *не в смежных блоках* памяти, в смежных блоках (вплотную друг к другу) *гарантированно* размещаются *только* компоненты одного и того же массива.

По указанной причине сравнение указателей операциями $>$, $<$, $>=$, $<=$ также имеет смысл только для указателей, направленных на компоненты одного и того же массива. Однако операции отношения $==$ и $!=$ правомерно использовать *для любых* указателей.

10.6. Массивы указателей

На предыдущих уроках в качестве элементов массивов использовались в основном данные простых типов (*int*, *float* и др.). На двенадцатом уроке в ячейки массива будем записывать текстовые переменные – вереницы символов. На седьмом уроке отмечалось, что многомерный массив можно представить в виде массива массивов. В частности, двумерный массив выражается в виде одномерного массива, в ячейках которого хранятся одномерные массивы одного и того же типа и *одинаковой* длины. Напомним ещё раз описание и инициализацию матрицы *M*, состоящей из *nSt* строк, и *nCl* столбцов:

```
const int nSt= 4, nCl= 2; //Число строк и столбцов
```

```
typedef int Matrica[nSt][nCl]; //Определение типа
```

```
Matrica M={{1, 1}, {2, 2}, {3, 3}, {4, 4}};
```

Как видите, в вышеприведенном описании значения элементов каждой строки размещаются в фигурных скобках, строки отделяются друг от друга, также как и элементы строки, при помощи запятых. Если увеличить число предварительно заказанных строк nSt и столбцов nCl , например в 10 раз (или сделать так, чтобы они определялись по умолчанию, с использованием пустых квадратных скобок), а инициализацию массива оставить прежней, то *Builder* при помощи внутренних фигурных скобок правильно разместит введенные числа по строкам и столбцам.

Инициализацию многомерного массива (в нашем случае двумерного) можно осуществить и без использования внутренних фигурных скобок:

```
Matrica M={1, 1, 2, 2, 3, 3, 4, 4};
```

В этом случае число строк и столбцов *не может* быть больше или меньше, чем указано в предшествующем инициализирующем выражении, не может определяться с использованием пустых квадратных скобок. В противном случае компоненты инициализирующего выражения окажутся размещёнными не по вашему желанию, поскольку *Builder* его просто не знает. При этом сообщение об ошибке может не появиться. Как отмечалось ранее (см. шестой урок), сообщение об ошибке выводится, когда число инициализирующих значений больше, чем заказанное количество элементов матрицы.

В памяти компьютера элементы матрицы размещаются так: вначале *непрерывным* блоком располагаются элементы первой строки, затем – второй строки и так далее вплоть до последней строки. Фактически матрицы (и любые иные многомерные массивы) *всегда* записываются в оперативной памяти в виде единого одномерного массива. Программист же *лишь для своего удобства* представляет этот массив в виде массива массивов или матрицы. Поэтому матрицу M можно описать ещё и так:

```
const int nSt= 4, nCl= 2;
typedef int Odnom_Massiv[nCl];
typedef Odnom_Massiv Matrica[nSt];
Matrica M={{1, 1}, {2, 2}, {3, 3}, {4, 4}};
```

Теперь в каждой ячейке одномерного массива $M[nSt]$, состоящего из nSt элементов, хранится также одномерный массив типа *Odnom_Massiv* из nCl компонентов (элементы строк).

В ячейках одномерного массива можно хранить не только одномерные массивы, но и *указатели* на одномерные массивы. В этом случае ячейки массива *распределяются* по оперативной памяти, строки и столбцы массива не представляют собой единый блок ячеек. Единым *непрерывным* блоком располагаются только адресные ячейки указателей на строки (каждая имеет объём в 4 байта), а объекты указателей строк могут не находиться в смежных ячейках памяти (однако все ячейки *каждой* строки, конечно, находятся в отдельных блоках друг к другу примыкающих ячеек).

Всё сказанное достаточно ясно, однако зачем применять массивы указателей, чем плох прежний способ размещения данных? – зададут читатели вполне резонный вопрос. Массив указателей на одномерные массивы используется для повышения быстродействия программ – ответим мы.

Описанная организация многомерных массивов позволяет выполнять их обработку (например, сортировку, какие-то другие изменения в расположении строк) *без перемещения* объектных частей указателей на строки. Порядок исходных строк (одномерных однотипных массивов одинаковой длины) легко изменяется только при помощи переустановки направленных на них указателей, сами же ячейки со значениями элементов одномерных массивов *остаются на прежних* местах. В этом случае существенно повышается скорость обработки, поскольку исключаются затраты времени на перезапись значений ячеек массива, на их перестановки. Перезапись адреса в адресную часть указателя осуществляется только *одной* операцией, вместе с тем перезапись всех элементов массива требует столько операций записи, сколько ячеек имеется в массиве. При этом требуется применение цикла, каждый шаг которого также увеличивает затраты времени. Прибегать к массивам указателей целесообразно в первую очередь при *длинных* массивах или *громоздких* структурах (структуры изучаются на тринадцатом уроке).

Ниже опишем работу приложения *Массив указателей*, в котором иллюстрируется работа с упомянутым типом данных. В приложении задаётся матрица M , состоящая из 4 строк и 2 столбцов. В файле реализации для размещения её элементов выделяется (с запасом) в 10 раз больше строк и столбцов, чем требуется по условию этой демонстрационной задачи ($nSt = 40, nCl = 20$). Пользовательская функция *Vvod_Matr*, вызываемая в функции *PyskClick*, осуществляет инициализацию ячеек матрицы M . Их значения выводятся на интерфейсе приложения (см. рис. 10.6) при помощи экземпляра объекта типа *TStringGrid* с именем *Tab_1*. Такой вывод значений матрицы M в ячейки упомянутой таблицы строк осуществляет пользовательская функция *Vuvod_Matr*.

В начале файла реализации, помимо матрицы M , объявляется также массив указателей *Mas_Yk* (из nSt элементов) на тип *int*. Далее в функции *PyskClick* при вызове функции *Vvod_Mas_Yk* в ячейки ранее объявленного массива указателей *Mas_Yk* записываются адреса строк матрицы M . Однако такая запись не может

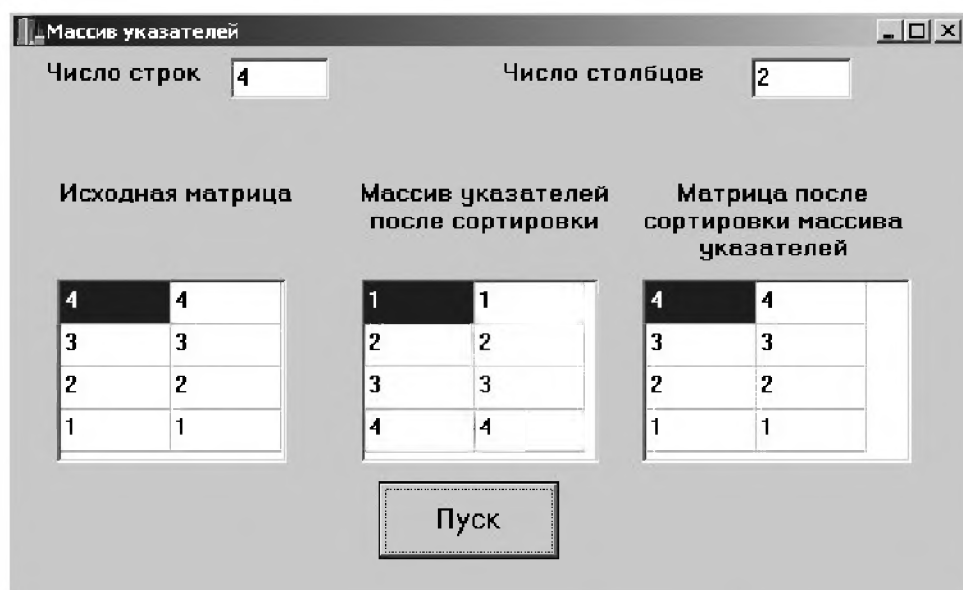


Рис. 10.6. Интерфейс приложения *Массив указателей*


```
Massiv Yk[ii]= (int *) &M[ii];
```

Функция *Vuvod_Mas_Yk* содержит все ячейки отсортированного массива указателей *Mas_Yk* выводит на интерфейс при помощи *второго* экземпляра объекта типа *TStringGrid* с именем *Tab_2*. Из интерфейса видно, что строки оказались отсортированными согласно упомянутому принципу (возрастанию суммы элементов строк), поэтому их расположение изменилось относительно строк исходной матрицы *M* и, следовательно, относительно исходного массива указателей *Mas_Yk*.

В приложении имеются также уже хорошо известные читателям поля ввода числа строк и столбцов матрицы, программные средства переустановки (по введенным значениям) числа строк и столбцов всех трёх объектов типа *TStringGrid*.

```
//Текст приложения Массив указателей
const int nSt= 40, nCl= 20;
typedef int Matrica[nSt][nCl];
Matrica M;
//Массив указателей
typedef int *Mas_Ykaz[nSt];
Mas_Ykaz Mas_Yk;

void Vvod_Matr(Matrica MM, int n_Strok, int n_Colon)
{
    for (int ii= 0; ii< n_Strok; ++ii)
        for (int ij= 0; ij< n_Colon; ++ij)
```

```

        MM[ii][ij]= n_Strok - ii;
    } //Vvod_Matr

void Vuvod_Matr(TStringGrid *Tabl, Matrica MM,
               int n_Strok, int n_Colon)
{
    for (int ii= 0; ii< n_Strok; ++ii)
        for (int ij= 0; ij< n_Colon; ++ij)
            Tabl->Cells[ij][ii]= MM[ii][ij];
    } //Vuvod_Matr

void Vvod_Mas_Yk(Mas_Ykaz Ma_Yk, int n_Strok)
{
    for (int ii= 0; ii< n_Strok; ++ii)
        Ma_Yk[ii]= (int *)&M[ii];
    } //Vvod_Mas_Yk

float Symma_Stroki(Mas_Ykaz Ma_Yk, int n_Colon, int Nomer_Str)
{
    float fS= 0;
    for (int ij= 0; ij< n_Colon; ++ij)
        fS += Ma_Yk[Nomer_Str][ij];
    return fS;
    } //Symma_Stroki

void Sortirovka_Strok(Mas_Ykaz Ma_Yk, int n_Strok, int n_Colon)
{
    int *iBuf;
    for (int ii= 0; ii< n_Strok - 1; ++ii)
        for (int iCan= ii+1; iCan< n_Strok; ++iCan)
            if (Symma_Stroki(Ma_Yk, n_Colon, ii )>
                Symma_Stroki(Ma_Yk, n_Colon, iCan))
                {
                    iBuf= Ma_Yk[ii];
                    Ma_Yk[ii]= Ma_Yk[iCan];
                    Mas_Yk[iCan]= iBuf;
                } //if
    } //Sortirovka_Strok

void Vuvod_Mas_Yk(TStringGrid *Tabl,
                 Mas_Ykaz Ma_Yk, int n_Strok, int n_Colon)
{
    for (int ii= 0; ii< n_Strok; ++ii)
        for (int ij= 0; ij< n_Colon; ++ij)
            Tabl->Cells[ij][ii]= Ma_Yk[ii][ij];
    } //Vuvod_Mas_Yk

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    const int n_St = StrToInt(Ch_Str->Text),
              n_Cl = StrToInt(Ch_Cl->Text);
    Tab_1->RowCount= n_St;
    Tab_1->ColCount= n_Cl;
    Tab_2->RowCount= n_St;
    Tab_2->ColCount= n_Cl;
    Tab_3->RowCount= n_St;

```

```

Tab_3->ColCount= n_Cl;
Vvod_Matr(M, n_St, n_Cl);
Vuvod_Matr(Tab_1, M, n_St, n_Cl);
Vvod_Mas_Yk(Mas_Yk, n_St);
Sortirovka_Strok(Mas_Yk, n_St, n_Cl);
Vuvod_Mas_Yk(Tab_2, Mas_Yk, n_St, n_Cl);
Vuvod_Matr(Tab_3, M, n_St, n_Cl);
} // PyskClick

```

10.7. Динамические массивы

10.7.1. Одномерные динамические массивы

Ниже приводится код, в котором указатели направляются (инициализируются) на объект простой переменной типа *int*, объект массива типа *int* и объект *динамического* массива типа *int*.

```

typedef int *Yk_int;
const int in= 4;
int ix= 12, iMas[in];
Yk_int Yk_ix= &ix, //Настроен на объект переменной ix
        Yk_iMas= &iMas, //Настроен на объект массива iMas
        Din_M= new int[in]; //Порождение динамического массива

```

Запомните синтаксис применения операции *new* при порождении *динамического массива* и объявления типа этого массива. Именно так рекомендуем выполнять объявления динамических массивов. Вместе с тем, допустимо тип динамического массива и его порождение объединять в одной строке кода программы:

```
int *Din_M= new int[in];
```

Последний вариант определения нам представляется менее ясным. Отмечаем, что впереди квадратных скобок (с числом элементов массива) допустимы один и более пробелов (однако не рекомендуем применять такой синтаксис).

Перед изучением динамических массивов вспомним, что собой представляют указатели на обычные переменные и массивы.

На рис. 10.7 показаны адресные и объектные части указателей, инициализированные целочисленной переменной (верхняя часть рисунка) и статическим целочисленным массивом. Как видно, обе части этих переменных-указателей расположены в сегменте данных, объект простой переменной состоит из одной ячейки, а объект указателя на массив – из четырёх ячеек. В каждой из ячеек можно хранить только данные типа *int*.

Ячейки объектной части динамического массива *Din_M* находятся в динамической памяти (куче), а его адресная часть расположена в сегменте данных (см. рис. 10.8).

Более подробно с одномерными *динамическими* массивами познакомимся в ходе изучения приложения *Одномерный динамический массив*. В начале файла реализации программы вводится пользовательский тип данных *Yk_int*:

```
typedef int *Yk_int;
```

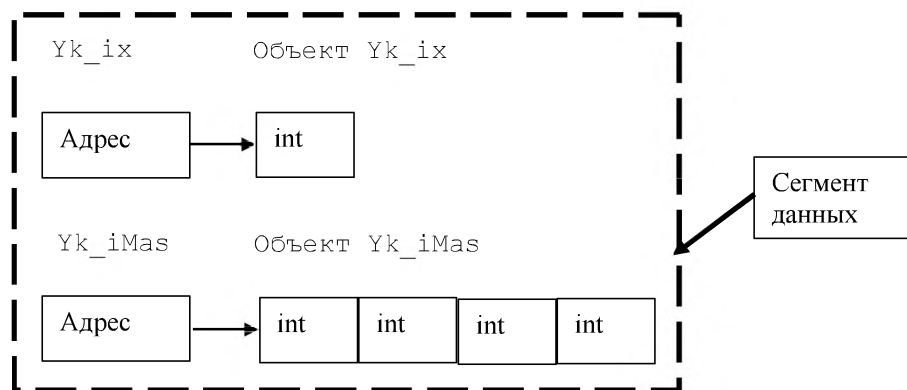


Рис. 10.7. Вид указателей на статическую переменную (верхняя часть рисунка) массив (нижняя часть рисунка)

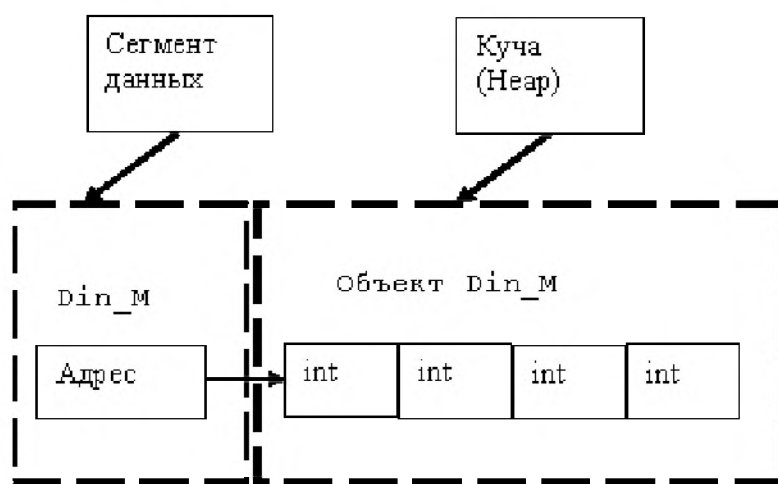


Рис. 10.8. Адресная и объектная части одномерного целочисленного динамического массива *Din_M*

Этот тип используется для передачи динамического массива в качестве параметра пользовательских функций *Vvod_Mas* и *Vuvod_Mas*. Первая из них предназначена для инициализации одномерного динамического массива *Din_M*, а вторая – для вывода его значений на интерфейс приложения при помощи таблицы строк *Tabl* (см. рис. 10.9).

Ясно, что вместо типа *int* может быть тип *float* или любой другой числовой тип данных (и не только числовой). Необходимо *особо* подчеркнуть, что при определении динамических массивов размер массива *in* можно задавать как константой и константным выражением (как в статических массивах), так и при помощи *инициализированной* переменной. При этом динамические массивы, в отличие от статических, можно порождать *в ходе выполнения программы*, и они необходимы для решения конкретного этапа обработки данных. Поэтому определение динамического массива возможно при помощи поля ввода с именем *Vvod_n* инициализировать переменную *in* – значение длины массива:

```
int in = StrToInt(Vvod_n->Text);
```

В случае статических массивов вышеприведенное выражение недопустимо.

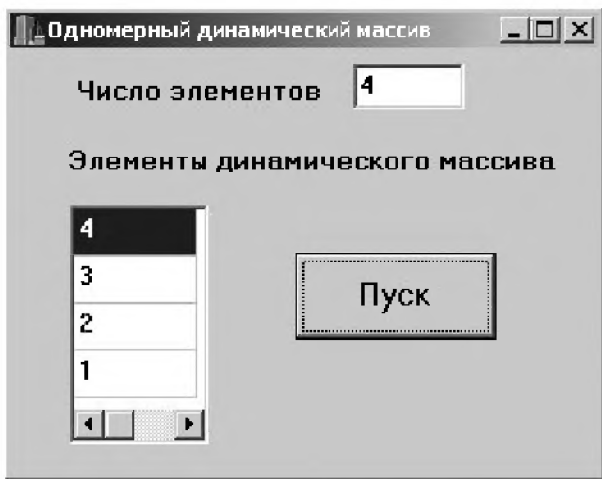


Рис. 10.9. Интерфейс приложения
Одномерный динамический массив

даже с применением спецификатора *const*:

```
const int in = StrToInt(Vvod_n->Text);
```

В теле функции *PyskClick* после ввода длины массива *in* и установки числа строк таблицы строк *Tabl* происходит порождение динамического массива и запись адреса его объекта (адреса начальной ячейки объекта) в переменную *Din_M* типа «указатель на *int*»:

```
Yk_int Din_M= new int[in];
```

При определении глобальных статических массивов они инициализируются нулевыми значениями, если это числовые массивы, и пустой строкой в случае массивов текстовых переменных. После порождения динамических массивов (как глобальных, так и локальных) значения их ячеек являются *неприсвоенными* (т.е. в них находится «мусор» — информация, оставшаяся от предшествующего использования ячеек памяти). Аналогичная ситуация наблюдается также и для значений локальных статических массивов (расположенных в стеке) после их объявления.

После инициализации и вывода значений динамического массива *Din_M* (в приложении в этом только и заключается его обработка) этот массив уничтожается. Процесс уничтожения разбивается на два этапа: вначале освобождаются ячейки памяти, использованные для размещения объектной части динамического массива, а затем в его адресную часть записывается константа *NULL*, позволяющая последующее корректное использование идентификатора *Din_M* (в приложении эта возможность не реализуется).

```
delete [] Din_M;  
Din_M= NULL;
```

Указанные операции являются признаком хорошего стиля программирования, в рассматриваемой программе без них, конечно же, можно обойтись. Однако настоятельно рекомендуем упомянутыми операциями *никогда* не пренебрегать. Отмечаем, что впереди и позади пустых квадратных скобок пробелы могут отсутствовать, может быть по одному или более пробелов. Далее применяется только синтаксис с одним пробелом.

```
//Файл реализации приложения Одномерный динамический массив
typedef int *Yk_int;

void Vvod_Massiva(Yk_int Mas, int n_Elem)
{
    for (int ii= 0; ii<n_Elem; ++ii)
        Mas[ii]= n_Elem - ii;
} //Vvod_Massiva

void Vuvod_Massiva(TStringGrid *Tab, Yk_int Mas, int n_Elem)
{
    for (int ii= 0; ii<n_Elem; ++ii)
        Tab->Cells[0][ii]= Mas[ii];
} //Vuvod_Massiva

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    //Вводим значение переменной, предназначенной для хранения
    //длины массива
    int in = StrToInt(Vvod_n->Text);
    //Изменяем число строк таблицы Tab1
    Tab1->RowCount= in;
    //Порождаем динамический массив и его адрес
    //записываем в указатель Din_M
    Yk_int Din_M= new int[in];
    //Менее предпочтительный способ порождения Din_M
    //int *Din_M= new int[in];
    //Инициализируем ячейки динамического массива
    Vvod_Massiva(Din_M, in);
    //Выводим значения динамического массива на экран
    Vuvod_Massiva(Tab1, Din_M, in);
    //Уничтожаем объект динамического массива
    delete [] Din_M;
    //В адресную часть Din_M записываем NULL или 0
    Din_M= NULL;
} //PyskClick
```

В заголовках пользовательских функций можно *не применять* пользовательские типы данных, объявленные при помощи служебного слова *typedef*. Если, например в функции *Vvod_Massiva*, не использовать введенный тип *Yk_int*, то её заголовок выглядит так:

```
void Vvod_Massiva(int *Mas, int n_Elem)
```

Нам представляется такой синтаксис менее ясным, поэтому в настоящем пособии он по возможности не используется.

10.7.2. Двумерные динамические массивы

Для изучения правил работы с многомерными динамическими массивами, рассмотрим приложение *Двумерный динамический массив*. По своим задачам эта программа аналогична приложению, рассмотренному в предшествующем пункте. Интерфейс приложения показан на рис. 10.10.

Работа с двумерными массивами начинается с определения типа данных «указатель на указатель на тип *int*» (или указатель на указатель на любой другой

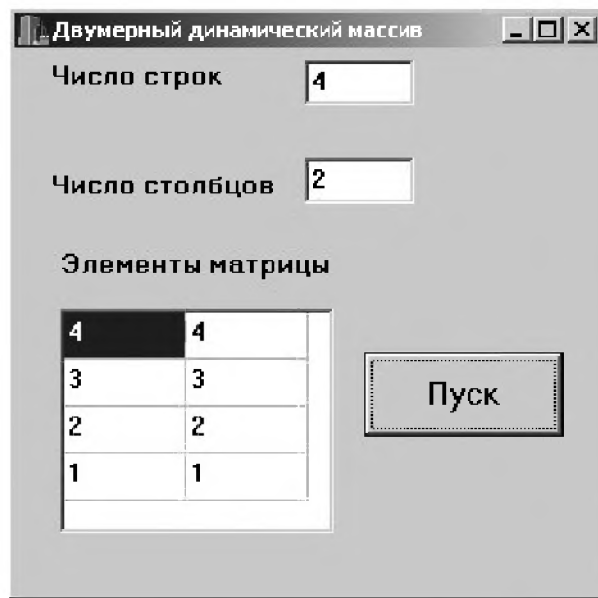


Рис. 10.10. Интерфейс приложения *Двумерный динамический массив*

тип данных).

```
typedef int **Yk_Yk_int;
```

Такое определение типа позволяет динамические матрицы данных передавать (с более наглядным синтаксисом) в функции *Vvod_Matr* и *Vuvod_Matr* в качестве их фактических параметров. Назначение этих функций состоит в инициализации матрицы целочисленных данных и выводе её значений на интерфейс в объект типа *TStringGrid* с именем *Tabl*.

Введенный тип данных *Yk_Yk_int* представляет собой, как правило, динамическую переменную, адресная часть которой находится в *сегменте данных*. Объектная часть этой переменной располагается в *динамической памяти* и состоит из массива *смежных* ячеек, занимающих *единый* блок памяти. В каждой из этих ячеек можно хранить *только адреса* ячеек выбранного типа (в нашем примере – типа *int*), которые также находятся в динамической памяти. Эти ячейки являются объектной частью своих адресных ячеек, обычно они *рассредоточены* в оперативной памяти (*могут находиться не в едином блоке*). На рис. 10.11 схематично изображена структура двумерного динамического массива *M_Din*.

Объявление типа *Yk_Yk_int* не приводит к порождению самой динамической переменной, однако этот тип используется при порождении двумерного динамического массива данных.

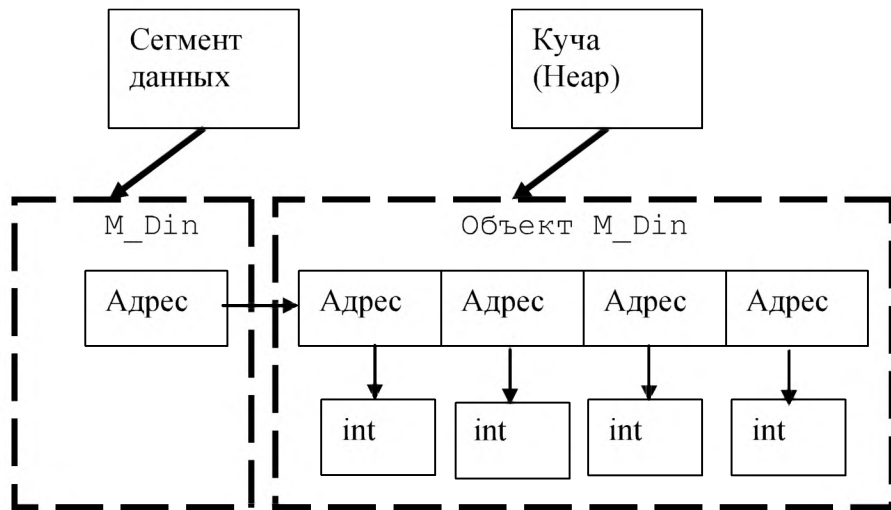
Процесс порождения динамической матрицы разбивается на два этапа. Вначале в переменную *M_Din* типа *Yk_Yk_int* записывается адрес массива указателей на строки матрицы типа *int*:

```
Yk_Yk_int M_Din= new int *[n_St];
```

Эту же операцию можно выполнить с использованием менее ясного синтаксиса:

```
int **M_Din = new int *[n_St];
```

После создания *массива* таких указателей в каждую его ячейку записывается



с. 10.11. Адресная и объектная части двумерного целочисленного динамического массива M_Din

рождаемого динамического массива (у каждой адресной ячейке – свой массив). Это оказывается возможным в результате повторного операции *new*, теперь уже в цикле *for*:

```
for (ii=0; ii< n_St; ii++)
    M_Din[ii] = new int[n_Cl];
```

Уничтожение двумерного динамического массива производится не так, как в случае одномерного динамического массива. Вначале следует освободить память, выделенную для размещения динамических массивов ($M_Din[ii]$), а затем вернуть память, использованную для динамического массива указателей на них (M_Din):

```
for (ii=0; ii< n_St; ii++)
    delete M_Din[ii];
delete M_Din;
```

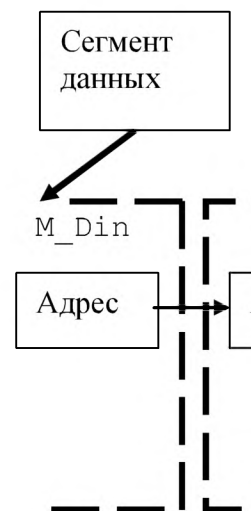
Таким образом, процесс уничтожения динамической матрицы противоположен процессу её порождения. Обращаем внимание на то, что при уничтожении динамической матрицы после второй операции *delete* следует ставить лишь одну открывающую и закрывающую квадратные скобки.

Реализация приложения Двумерный динамический массив

```
// Указатель на указатель на тип int
int **Ykaz_na_Ykaz_na_int;

// Функция создания матрицы
void CreateMatr(Yk_Yk_int MM, int n_Strok, int n_Colon)
{
    for (int ii= 0; ii< n_Strok; ++ii)
        for (int ij= 0; ij< n_Colon; ++ij)
            MM[ii][ij]= n_Strok - ii;
}

// Функция освобождения матрицы
void FreeMatr(TStringGrid *Tab, Yk_Yk_int MM, int n_Strok, int n_Colon)
{
    for (int ii= 0; ii< n_Strok; ++ii)
```




```

    for (int ij= 0; ij< n_Colon; ++ij)
        Tab->Cells[ij][ii]= MM[ii][ij];
} //Vuvod_Matr

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    //Ввод числа строк и столбцов матрицы
    int n_St = StrToInt(Ch_Str->Text),
        n_Cl = StrToInt(Ch_Cl->Text);
    //Задание числа строк и столбцов таблицы Tab1
    Tab1->RowCount= n_St;
    Tab1->ColCount= n_Cl;
    //Определение двумерного динамического массива
    //В адресную часть M_Din записываем адрес
    //на динамический массив указателей на int
    Yk_Yk_int M_Din= new int *[n_St];
    //или int **M_Din = new int *[n_St];
    for (int ii=0; ii< n_St; ii++)
        M_Din[ii]= new int[n_Cl];
    Vvod_Matr(M_Din, n_St, n_Cl);
    Vuvod_Matr(Tab1, M_Din, n_St, n_Cl);
    //Уничтожение двумерного динамического массива
    for (int ii=0; ii<n_St; ii++)
        delete M_Din[ii];
    delete [] M_Din;
    M_Din= NULL;
} //PyskClick

```

Напоминаем, заголовок функции, например, *Vvod_Matr* можно оформить и так:

```
void Vvod_Matr(int ** MM, int n_Strok, int n_Colon)
```

10.8. Указатели на функции

Функцию можно передавать через указатель на неё. Для этого объявляется указатель на функцию и ему при помощи операции взятия адреса присваивается адрес функции (это один из вариантов такого присваивания). Для знакомства с синтаксисом этих операций рассмотрим приложение *Указатель на функцию*, в котором в полях ввода *Vvod_X* и *Vvod_Y* вводятся числа, сумма которых появляется на экране после нажатия кнопки *Пуск*. Интерфейс приложения показан на рис. 10.12, а файл реализации приведен ниже.

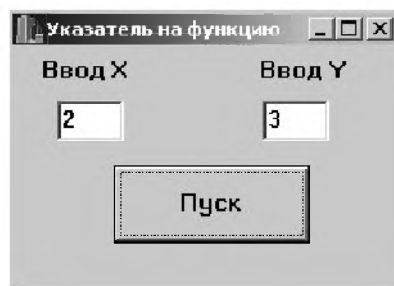


Рис. 10.12. Интерфейс приложения *Указатель на функцию*

```
//Файл реализации приложения Указатель на функцию
```

```
...
```

```
typedef int (*Yk_Fyn)(int ixx, int iyy);
```

```
int Symma(int ixx, int iyy)
```

```
{
    int iz= ixx +iyy;
    return iz;
} //Symma
```

```
void Pokaz(Yk_Fyn Fynkc, int ixx, int iyy)
```

```
{
    ShowMessage(Fynkc(ixx, iyy));
    //или ShowMessage((*Fynkc)(ixx, iyy));
} // Pokaz
```

```
void __fastcall TForm1::PyskClick(TObject *Sender)
```

```
{
    int ix= StrToInt(Vvod_X->Text),
        iy= StrToInt(Vvod_Y->Text);
    //Указателю Ykaz_na_Fun присваивается адрес функции
    Yk_Fyn Ykaz_na_Fun= &Symma;
    //или Yk_Fyn Ykaz_na_Fun= Symma;
    //Функция вызывается через указатель
    ShowMessage(Ykaz_na_Fun(ix, iy));
    ShowMessage((*Ykaz_na_Fun)(ix, iy));
    Pokaz(&Symma, ix, iy);
} //PyskClick
```

После нажатия кнопки *Пуск* для параметров, введенных на интерфейсе, функция *ShowMessage* выведет трижды значение 5. Обращаем внимание на синтаксис задания указателя на функцию: имя указателя *вместе со звёздочкой* (слева от указателя) располагаются *внутри* круглых скобок, типы параметров функции перечисляются через запятую *внутри* других круглых скобок, имена формальных параметров при этом можно не указывать.

```
int (*Yk_Fun)(int ixx, int iyy);
```

Тип возвращаемого указателем значения, количество параметров указателя и последовательность их типов в списке вызова должны *полностью* совпадать с соответствующими характеристиками тех функций, которыми предполагается инициализировать описываемый указатель. Во всех последующих примерах на это *очевидное* требование более *не будем* обращать внимание.

Вызов указателя на функцию ничем не отличается от вызова функции:

```
ShowMessage(Ykaz_na_Fun(ix, iy));
```

Таким образом, в данном случае *Ykaz_na_Fun* является синонимом функции *Symma*. Предшествующий вызов указателя можно осуществить ещё и так:

```
ShowMessage((*Ykaz_na_Fun)(ix, iy));
```

Здесь *(*Ykaz_na_Fun)* является операцией разыменования указателя: обращение к ячейке, начиная с которой записан код функции, вызываемой с параметрами, приведенными в последующих, вторых, круглых скобках. Как видите, эта

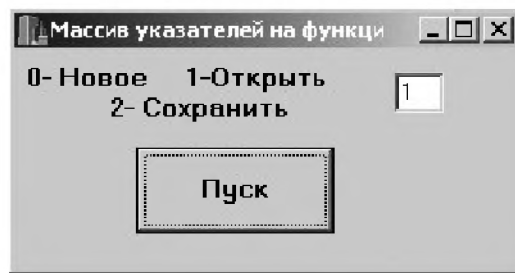


Рис. 10.13. Интерфейс приложения *Массив указателей на функции*

форма вызова также вполне логична, однако нам она представляется менее привлекательной, чем первый вариант использования указателя на функцию.

Можно объявить *массивы* указателей на *целый ряд* функций. К примеру, это бывает полезным при разработке меню приложения. Покажем реализацию такой возможности на примере приложения *Массив указателей на функции*, его интерфейс показан на рис. 10.13. После ввода в поле ввода *Vvod* выбранного значения, на экран выводится соответствующее слово.

//Файл реализации приложения Массив указателей на функции

```
void Novoe(void) //void внутри скобок может отсутствовать
{
    ShowMessage("Новое");
} //Novoe
```

```
void Otkrutj(void)
{
    ShowMessage("Открыть");
} //Otkrutj
```

```
void Soxranitj(void)
{
    ShowMessage("Сохранить");
} //Soxranitj
```

```
void __fastcall TForm1::PyskClick(TObject *Sender)
{
    int ix= StrToInt(Vvod->Text);
    if (ix<0 || ix>2)
    {
        ShowMessage("Введено число вне диапазона");
        Abort();
    } //if
    //Описание типа указателя на функцию без параметра
    typedef void (*Tip_Menu)(void);
    //Описание и инициализация массива указателей Menu[3]
    //на функции без параметров
    Tip_Menu Menu[3]= {&Novoe, &Otkrutj, &Soxranitj};
    //Вызов функции
    Menu[ix]();
    //или (*Menu[ix])();
} //PyskClick
```

Синтаксис инициализации массива указателей аналогичен инициализации обычного массива, поэтому вполне логичен. Вызов необходимого указателя так-

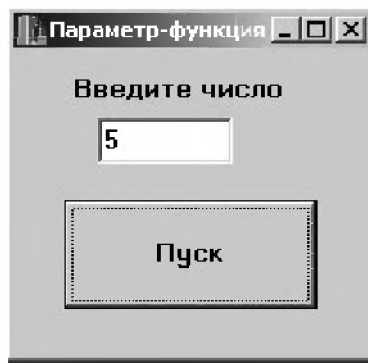


Рис. 10.14. Интерфейс приложения *Параметр-функция*

же достаточно понятен: пустые круглые скобки показывают, что вызываются указатели на функции, а не указатели на данные. Если в вызываемых функциях будет один или серия параметров, то их следует перечислять в той же последовательности в круглых скобках, которые в нашем приложении остались пустыми. Также как и в предшествующем примере, вызов указателя на одну из функций меню возможен через операцию разыменования.

Видимо, полная ясность в вопросе об использовании указателей на функции наступит после рассмотрения приложения *Параметр-функция*, его интерфейс приведен на рис. 10.14. Приложение выводит на экран квадрат числа, введенного в поле ввода *Vvod_Chisla*. Просмотрите код приложения и комментарии к нему.

```
//Файл реализации приложения Параметр-функция
typedef int (*Yk_Fyn)(int Chslo);
//Функция Pokaz получает в качестве параметра
//указатель Fynkc типа Yk_Fyn

void Pokaz(Yk_Fyn Fynkc, int ixx)
{
    ShowMessage(Fynkc(ixx));
    //или ShowMessage((*Fynkc)(ixx));
} //Pokaz

int Kvadrat(int Chslo)
{
    int ix= Chslo*Chslo;
    return ix;
} //Kvadrat

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    int ix= StrToInt(Vvod_Chisla->Text);
    Pokaz(Kvadrat, ix);
} //PyskClick
```

Важно отметить, что единственный параметр функции *Kvadrat* передаётся в пользовательскую функцию *Pokaz*, как её *второй* аргумент, *первым* же аргументом функции *Pokaz* является функция *Kvadrat*, поскольку в имени функции (в нашем примере в функции *Kvadrat*) хранится адрес начальной ячейки исполняемого кода функции (указатель на тип *Yk_Fyn* упомянутой функции):

```
void Pokaz(Yk_Fyn Fynkc, int ixx)
```

Это единственный новый элемент синтаксиса, который необходимо знать для использования указателей на функцию. Указатели на функции незаменимы, когда разрабатывается программа решения задачи, в которой могут использоваться различные методы вычислений, реализованные разными функциями.

Например, в каком-то приложении в функции *Fynkc_1* описывается одна из операций численного интегрирования произвольной зависимости, а в функции *Fynkc_2* эта операция применяется в ходе каких-то вычислений для определения интегралов нескольких конкретных зависимостей. Так вот, в функцию *Fynkc_2* можно передать в качестве формального параметра указатель *Yk_Fynkc* на функцию *Fynkc_1* вместе с её формальными параметрами (границами интервала, задаваемым шагом интегрирования). В ходе разработки этого же приложения можно запрограммировать функцию *Fynkc_3*, в которой интегрирование выполняется иным методом, чем в функции *Fynkc_1*. В этом случае при вызове *прежней* (неизменной) функции *Fynkc_2* ей следует передать лишь имя *Fynkc_3* вместо *Fynkc_1*. При этом конечно, типы функций *Fynkc_3* и *Fynkc_1* должны быть тождественны. Аналогичный пример: нахождение корней уравнения, где могут применяться различные методы вычисления. В приложении *Задача 10.2* (см. раздел 10.11) рассматривается именно такой вариант применения указателя на функцию (непреренно изучите это приложение).

10.9. Функции, возвращающие указатель на массив

Как разрабатывать функции, возвращающие указатель на массив, покажем на следующем примере. В приложении задаётся целочисленный массив из 4 чисел. В функции *PyskClick* вызов пользовательской функция *Sort_Mas* упорядочивает этот массив по возрастанию значений его элементов (методом линейной сортировки). Однако вывод значений отсортированного массива осуществляется не при помощи идентификатора массива, а с использованием указателя на тип массива *int*, который возвращает функция *Sort_Mas*. Для последовательного вывода упомянутых значений массива применяется функция *ShowMessage* в цикле *for* по элементам массива. Прочтите код приложения и его комментарии.

```
//Файл реализации приложения функция, возвращающая указатель на массив
...
const int n= 4;
//Пользовательский тип на массив типа int из n элементов
typedef int int_Mas[n];
//Объявляем и инициализируем массив типа int_Mas
int_Mas iMas={8, 6, 2004, 54};
//Объявляем тип «указатель на массив типа int»
typedef int *Yk_int;
//Объявляем переменную Yk_Mas типа Yk_int
Yk_int Yk_Mas;

//Функция, которая возвращает указатель на массив
Yk_int Sort_Mas(int_Mas iMa, int iDlina)
```

```
//Осуществляем линейную сортировку
{
    for (int ij= 0; ij< iDlina-1; ++ij)
        for (int iCan= ij + 1,iByf; iCan< iDlina; ++iCan)
            if (iMa[ij]> iMa[iCan])
            {
                iByf= iMa[ij];
                iMa[ij]= iMa[iCan];
                iMa[iCan]= iByf;
            }//if
    Yk_Mas= iMa;
    return Yk_Mas;
} //Sort_Mas

void __fastcall TVozvrash_Yk_na_Int::PyskClick(TObject *Sender)
{
    Sort_Mas(iMas, n);
    for (int ii= 0; ii< n; ++ii)
        ShowMessage(Yk_Mas[ii]);
} //PyskClick
```

Обращаем внимание на то, что указатель следует объявлять на тип *int*, т. е. на тип элементов, которые должны храниться в заданном массиве. Если тип исходного массива был бы, например, *float*, то и указатель должен объявляться на тип *float*. Фактически объявляется указатель не на массив, а на тип его ячеек, вместе с тем при помощи такого указателя возможно передать из функции *все* элементы массива.

Зачем это нужно, спросит дотошный читатель, — ведь в пользовательские функции массивы *всегда* передаются *только* по адресу (а не по значению), поэтому для использования изменённых (преобразованных) элементов массива в любых местах программы нет никаких проблем? Да, это так. Однако если сама пользовательская функция формирует массив данных, то их разумно передать и таким экзотическим способом. Впрочем, удобнее это сделать всё же через параметр-массив. В общем, в *C++* имеется и такая возможность передачи массива, и вы теперь ею владеете, а в бою разнообразные арсеналы вооружений всегда могут оказаться полезными.

10.10. Важная рекомендация

Дорогие читатели, сейчас, когда вы изучили богатые возможности указателей и приобрели практические навыки работы с ними, познакомьтесь с рекомендацией профессионалов: в *C++ Builder* не принято объявлять указатели на какие-либо типы данных. В приложениях применяйте указатели *только* на *стандартные* типы, использовать объекты которых по иному просто *невозможно*. К таким типам объектов относятся, например, все объекты из библиотеки визуальных компонентов, многие объекты библиотечных классов и модулей. Однако *не расстраивайтесь* из-за впустую затраченных на первый взгляд времени и сил, ведь и с уже существующими указателями необходимо оперировать *осознанно*, полностью понимая, что они собой представляют.

На этом уроке вы убедились, что с *пользовательскими* указателями работать всё же *разрешено*, что их применение действительно в некоторых случаях очень эффективно. Почему же рекомендуется ими *не злоупотреблять*? Такая рекомендация обусловлена слишком *большой ответственностью*, которая ложится на программиста при обращении к тем или иным ячейкам памяти (особенно при записи в них данных), ведь эти ячейки могут использоваться операционной системой или стандартными функциями самого же *Builder*. У *Builder* имеются практически неограниченные возможности по разработке приложений с использованием различных видов вооружений его многочисленного мирного арсенала. Поэтому упомянутое ограничение не так уж и ущемляет возможности программиста, зато *существенно повышает надёжность* приложений, что в конечном итоге является самым главным при их разработке.

Вопросы для самоконтроля

- ☐ Что собой представляет статическая и динамическая память, статические и динамические переменные?
- ☐ Назовите операции порождения и уничтожения динамических переменных.
- ☐ Зачем применяется сегмент данных?
- ☐ Как узнать адрес переменной?
- ☐ Укажите способ доступа к объектам статических и динамических переменных?
- ☐ С какой целью используются неконстантный указатель на константные данные, константный указатель на неконстантные данные и константный указатель на константные данные?

10.11. Задачи для программирования

Задача 10.1. Осуществите обмен значений двух переменных с использованием указателя, инициализированного динамической переменной.

Задача 10.2. Разработайте программу, которая находит корни уравнения методом половинного деления области определения (метод дихотомии). Пусть уравнение $x^2 + x - 6 = 0$ задано в интервале $x \in [-10, 10]$. Используйте указатель для передачи функции поиска корней уравнения на заданном интервале монотонности зависимости $y = x^2 + x - 6$ в функцию поиска решения на всех интервалах монотонности.

10.12. Варианты решений задач

Решение задачи 10.1

```
void __fastcall TObmen::PyskClick(TObject *Sender)
{
    int ix= 55, iy= 54;
```

```

int *Yk= new int;
*Yk= ix;
ix= iy;
iy= *Yk;
ShowMessage("x= " + IntToStr(ix)+ " y= " + IntToStr(iy));
//Покажется сообщение "x= 54 y= 55".
} //PyskClick

```

Решение задачи 10.2

```

//fAA, fBB, fdxx - границы интервала и погрешность
typedef float (*Yk_Fyn)(float fAA, float fBB, float fdxx);

//Уравнение, корень которого необходимо вычислить
float fy(float fxx)
{
    float fyy= pow(fxx, 2) + fxx - 6;
    return fyy;
} //корни: x1= -3, x2= 2
} //fy

//Метод дихотомии - половинного деления области определения
float Korenj(float fAA, float fBB, float fdxx)
{
    float fyy;
    if (fy(fAA)<0)
        while (fBB - fAA> fdxx)
        {
            if (fy((fAA + fBB)/2)> 0.0)
                fBB= (fAA + fBB)/2;
            else
                fAA= (fAA + fBB)/2;
        } // while_1
    else
        while (fBB - fAA> fdxx)
        {
            if (fy((fAA + fBB)/2)< 0.0)
                fBB= (fAA + fBB)/2;
            else
                fAA= (fAA + fBB)/2;
        } // while_2
    fyy= (fAA + fBB)/2;
    return fyy;
} //Korenj

//nn - число интервалов монотонности, на которые
//разбивается область определения уравнения
void Opredelenie_Korenej(Yk_Fyn Yk_na_Fynk,
                        float fAA, float fBB, float fdxx, int nn)
{
    int iS= 0; //Счётчик числа корней
    float fA, fB, fh; //Переменные границы и шаг приращения
    //Величина интервала монотонности
    fh= (fBB - fAA)/nn;
    for(int ii=1; ii<= nn; ii++)

```



```
{
    fA= fAA + (ii - 1)*fh;
    fB= fAA + ii*fh;
    if (fy(fA)*fy(fB)<0)
    {
        iS++; // Номер корня
        ShowMessage( "x" + IntToStr(iS)+ "= " +
FloatToStrF(Yk_na_Fynk(fA, fB, fdxx), ffFixed, 10, 2));
    } //if
} //for_ii
if (iS==0)
    ShowMessage("Уравнение не имеет корней в интервале от " +
FloatToStrF(fAA, ffFixed, 10, 2) + " до " + FloatToStrF(fBB, ffFixed, 10, 2));
} //Opredelenie_Korenej

void __fastcall TReshen_Yravn::PyskClick(TObject *Sender)
{
    const float dx= 0.01, //Погрешность решения
        A= -10, B= 10, //Границы интервала
        n= 4; //Число отрезков монотонности
    Opredelenie_Korenej(Korenj, A, B, dx, n);
} // PyskClick
```



Урок 11. Ссылки

На этом и предшествующих уроках язык *C++* и возможности визуального программирования, реализованные *C++ Builder*, изучаются параллельно. Настоящий урок посвящён ссылочному типу данных и новым для вас объектам интерфейса.

11.1. Основные свойства

Ссылки упоминались ранее в основном при передаче параметров в функцию (см. восьмой урок). Передача параметров функции по ссылке — это, пожалуй, *наиболее важное* применение ссылок. Ниже рассмотрим другие возможности данных ссылочного типа. Приведём пример их объявления и использования:

```
int iX= 54;  
int &Ssul_int= iX;//Порождена переменная ссылочного типа  
ShowMessage(Ssul_int);//54
```

Функция *ShowMessage* выведет на экран число 54. Ссылка объявляется при помощи знака амперсанта, тип ссылки должен быть *тождественен* типу инициализирующего выражения. Впереди и после знака амперсанта *разрешается* ставить один и более пробелов, пробелы могут и отсутствовать.

Допустим и такой синтаксис инициализации ссылок:

```
int iX;  
int & Ssul_int= iX; iX= 54;
```

Его правомерность обусловлена правилом чтения операторов строки (разделённых точкой с запятой): чтение осуществляется справа налево. Однако не рекомендуем злоупотреблять такой формой инициализации, поскольку она существенно затрудняет беглое чтение программы.

Инициализировать ссылку можно ещё и так:

```
int &Ssul_int (iX);  
int &Ssul_int (iX= 55);
```

В целом ссылка используется для того, чтобы переменной (в том числе и объекту в смысле экземпляра класса) присвоить *ещё одно* имя или даже серию *дополнительных* имён. Поэтому ссылки при своём объявлении *обязательно* должны инициализироваться (как и константы). При этом переменная, которой инициализируется ссылка, может быть *неинициализированной*. Настроив ссылку на конкретную переменную, к объекту этой переменной можно обращаться двумя способами: по-старому — через имя переменной и по-новому — при помощи идентификатора ссылки. После инициализации *невозможно* разорвать установленную связь имени ссылки с объектом, на которой она была настроена, ещё говорят: *на который она смотрит*.

При помощи ссылки можно не только читать значение, записанное в объекте переменной, но и записывать в него другие допустимые значения:

```
int iX= 54, iY= 27;
int &Ssul_int= iX;
Ssul_int= iY;
ShowMessage(Ssul_int); //27
ShowMessage(iX); //27
```

В этом случае на экран дважды выводится число 27, что подтверждает возможность записи допустимых значений в объект переменной (в примере *iX*) при помощи ссылки на неё. Если третью строку предшествующего кода «*Ssul_int= iY;*» заменить строкой «*Ssul_int= 12;*», то на экране дважды появится число 12. Этим иллюстрируется *возможность* записи в объект переменной (*iX*) через ссылку на неё значения, которое не имеет адреса в памяти компьютера. Таким образом, однажды инициализированная ссылка *всегда* смотрит на объект инициализирующей переменной и все операции записи и чтения для ссылки *эквивалентны* выполнению этих же операций для имени инициализирующей переменной. Разрешается *неограниченное* число ссылок на переменную или создание *неограниченного числа синонимов*, псевдонимов, других имён переменной. При этом все эти синонимы *смотрят* на единственный объект – объект переменной, которой они инициализированы.

Применив к ссылке операцию взятия адреса &, получим не адрес ссылки, а адрес того объекта переменной, которой инициализирована ссылка. Поэтому, какие бы операции со ссылками не выполнялись – во всех случаях эти операции производятся с объектом указанной переменной. Как видите, ссылка *полностью эквивалентна* переменной, которая применяется для её инициализации. В частности, операция *sizeof* для ссылки определяет объём *переменной*:

```
double dX= 54.4861;
double &Ssul_double= dX;
ShowMessage(sizeof(Ssul_double)); //8
```

На экране покажется число 8, а не 4, как было бы в случае указателя.

Можно определить указатель на ссылку:

```
int iX= 54;
int & Ssul_int= iX;
int * Ykaz_Ssul= &Ssul_int;
ShowMessage(*Ykaz_Ssul); //54
```

В этом случае в адресную часть указателя *Ykaz_Ssul* записывается адрес объекта переменной *iX*.

Рассмотрим случай, когда тип ссылки *не тождественен* типу инициализирующего выражения, а лишь согласован с ним (тип инициализирующего выражения вложен в тип ссылки):

```
int iX= 54, iY= 27;
float &Ssul_float= iX; //или double, long double
Ssul_float= iY;
ShowMessage(Ssul_float); //27
ShowMessage(iX); //54
```

В этом случае не будет выведено сообщение об ошибке, а лишь появится пре-

дупреждение «*Temporary used to initialize 'Ssul_float'.*». Оно означает, что *Builder* по своей инициативе ввёл временную переменную с именем *Ssul_float*, которая не обладает свойством ссылок: на экране появится вначале число 27, а затем – 54. Это же предупреждение и число 27 появятся на экране, если ссылку инициализировать не переменной соответствующего типа, а значением, не имеющим адреса в памяти компьютера:

```
int &Ssul_int= 27;
ShowMessage(Ssul_int); //27
```

Ссылка может *смотреть* или *быть направлена только* на объект переменной, её *нельзя инициализировать* значением, которое ещё *не имеет адреса* в оперативной памяти компьютера. Поэтому в стандарте C++ приведенная выше операция *запрещена*. C++*Builder* выполняет эту операцию, воспринимая её по-иному (соответствующее предупреждение выводится в ходе компиляции).

Перечислим ограничения при определении и использовании ссылок:

1. Ссылка не может инициализироваться переменной с типом *void*.
2. Ссылку нельзя инициализироваться операцией *new*.
3. Не определены ссылки на другие ссылки.
4. Невозможно создать массивы ссылок.

Для закрепления указанных ограничений приведём ошибочные определения ссылок, в комментариях приведены сообщения об ошибках и номера ограничений, из-за которых эти описания являются неправильными:

```
int iX1= 54, iX2= 27, iX3= 56;
//void & is not a valid type
void & Ssul_void= iX1; //№1
//Cannot convert 'int *' to 'int'
int & Ssul_new= new int ; //№2
//Cannot define a pointer or reference to a reference
int & & Ssul_Ssul= iX1; //№3
//Array of references is not allowed
int & Ssul_Mas[3]={iX1, iX2, iX3}; //№4
```

Дополнительно к рассмотренным, приведём разрешённые операции для переменной ссылочного типа:

```
int & Ssul_new= *new int(2005);
ShowMessage(Ssul_new); //2005
```

В этом случае ссылка инициализируется адресом объекта безымянной динамической переменной, расположенного в куче.

```
int iX= 5;
int *Yk= &iX;
int &Ssul= *Yk;
ShowMessage(Ssul); //5
```

Здесь ссылка инициализируется адресом объекта целочисленной переменной, расположенного в сегменте данных. Эти же операции можно записать в более компактном виде:

```
int iX= 5, *Yk= &iX, &Ssul= *Yk; //Чтение слева направо
ShowMessage(Ssul); //5
```

11.2. Использование модификатора *const*

Разрешены ссылки на константы:

```
double dx= 3.141593; //Определена переменная
//Ссылка на постоянное значение
const double & Ssul_con_doub= dx;
```

Теперь значение ссылки *Ssul_con_doub* невозможно изменить, ей запрещено присвоение даже значения прежней переменной:

```
Ssul_con_doub= dx; //Cannot modify a const object
```

Не разрешается также и такое присваивание:

```
Ssul_con_doub= 55.5555; //Cannot modify a const object
```

В двух последних случаях на этапе компиляции программы выводится сообщение об ошибке *Cannot modify a const object* (нельзя модифицировать константный объект).

При определении ссылки на константу в качестве инициализатора можно использовать константное выражение или значение, ещё не имеющее адреса в памяти компьютера:

```
const double & Ssul_con_doub= 3.141593;
```

При этом тип этого значения или выражения может быть не тождественен типу ссылки. В следующем примере на стадии компиляции производится *автоматическое* преобразование типа (*int*) правостороннего выражения к типу ссылки (*double*):

```
const double & Ssul_con_doub= 2 + 1;
```

В обоих указанных вариантах в *Builder 5* выводится сообщение «*Temporary used to initialize 'Ssul_con_doub'*», которое следует игнорировать, поскольку на самом деле будет создана *полноценная* константная ссылка. В *Builder 6* такое предупреждение не выводится. Не рекомендуем использовать такие назначения для константных ссылок.

Некоторым аналогом константной ссылки является константный указатель на константные данные. Однако такой указатель на все типы данных (кроме массивов) при использовании его объекта необходимо разыменовывать, константную же ссылку разыменовывать не нужно. Массивы, по умолчанию обладают свойствами константных указателей, разыменовывать которые при применении их объекта также не требуется.

11.3. Ссылка на функцию

Ссылка на функцию описывается почти так же, как и указатель на функцию, только звёздочку следует заменить амперсантом. Синтаксис работы со ссылкой на функцию проиллюстрируем при помощи модернизации приложения *Параметр-функция*, рассмотренного на прошлом уроке (интерфейс задачи показан на рис. 10.14).

```
//Файл реализации приложения Параметр-функция
typedef int (& Ssul_Fyn)(int iChslo);

void Pokaz(Ssul_Fyn Fyn, int ixx)
{
    ShowMessage(Fyn(ixx));
} //Pokaz
int Kvadrat(int iChslo)
{
    int ix= iChslo*iChslo;
    return ix;
} //Kvadrat

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    int ix= StrToInt(Vvod_Chisla->Text); //5
    //Функция Pokaz получает в качестве параметра
    //ссылку типа Ssul_Fyn
    Pokaz(Kvadrat, ix); //25
} //PyskClick
```

В окне функции *ShowMessage* появится число 25 – квадрат введенного на интерфейсе числа 5.

В теле функции *PyskClick* можно определить переменную *Ssul_na_Fyn* типа *Ssul_Fyn*, а затем, после инициализации, использовать её идентификатор при вызове в функции *Pokaz*:

```
Ssul_Fyn Ssul_na_Fyn= Kvadrat; //Записывается адрес функции
Pokaz(Ssul_na_Fyn, ix); //Воспринимается как адрес функции
```

В этом примере имя функции *Kvadrat* и ссылка на неё *Ssul_na_Fyn* используются *без* скобок (и без параметров), поэтому они воспринимаются как *адрес* функции *Kvadrat*.

Инициализировать ссылку на функцию можно как при помощи знака присваивания (см. выше), так и круглых скобок:

```
Ssul_Fyn Ssul_na_Fyn(Kvadrat);
```

Можно инициализацию и объявление ссылки на функцию выполнить в одной строке, однако такая форма менее привлекательна:

```
int (& Ssul_na_Fyn)(int iChslo) = Kvadrat;
int (& Ssul_na_Fyn)(int iChslo) (Kvadrat);
```

Ссылка на функцию обладает *всеми* правами инициализирующей функции, является её синоним (псевдонимом). Вот так, например, можно вызвать функцию *Kvadrat* в функции *PyskClick*:

```
ShowMessage(Ssul_na_Fyn(ix));
```

В отличие от указателя на функцию, изменить значение ссылки на функцию *невозможно*. Видимо поэтому сфера применения ссылки на функцию значительно уже, чем указателя на функцию.

11.4. Функции, возвращающие ссылку

Функция может возвращать переменную ссылочного типа (или просто ссылку). Познакомимся с синтаксисом объявления и применения таких функций на примере приложения *Функция, возвращающая ссылку*. В программе объявляется и инициализируется массив *iMas* целочисленных значений, описывается функция *Max_El_Mas*, возвращающая ссылку на *максимальный* элемент массива, который передаётся ей в виде формального параметра *iMa*. Эта пользовательская функция применяется в функции *PyskClick* для вывода на экран максимального элемента 2004 массива *iMas* (это её фактический параметр).

Как отмечалось на восьмом и десятом уроках, все массивы – это константные *указатели* на неконстантные данные. Поэтому возвращаемый функцией *Max_El_Mas* элемент массива *iMa[iNom_Max]* (он стоит после оператора *return*) является *указателем*, перемещённым от начала массива *iMa* на количество *iNom_Max* компонент. Поэтому он представляет собой *адрес* ячейки, который вполне *правомерно* возратить функцией *Max_El_Mas*, поскольку возвращаемый ею её тип данных (*int &*) не что иное, как *адрес* ячейки с данными типа *int*.

При следующем вызове этой функции *Max_El_Mas* (в функции *PyskClick*) по указанному адресу записывается число 2005: *iMa[2] = 2005*. В результате при выводе на экран *всех* элементов фактического массива *iMas* на месте значения 2004 окажется число 2005.

```
//Файл реализации приложения функция, возвращающая ссылку
const int n= 4;
typedef int int_Mas[n];
int_Mas iMas={8, 6, 2004, 54};
//Функция, возвращающая ссылку на максимальный элемент массива
int & Max_El_Mas(int_Mas iMa, int iDlina)
{
    int iNom_Max= 0;
    for (int ij= 1; ij< iDlina; ++ij)
        if (iMa[ij]> iMa[iNom_Max])
            iNom_Max= ij;
    return iMa[iNom_Max];
} //Max_El_Mas

void __fastcall TVozvr_Ssul_Int::PyskClick(TObject *Sender)
{
    ShowMessage(Max_El_Mas(iMas, n)); //2004
    Max_El_Mas(iMas, n)= 2005; //Вместо 2004 записано 2005
    for (int ii= 0; ii< n; ++ii)
        ShowMessage(iMas[ii]); //8, 6, 2005, 54
} //PyskClick
```

11.5. Ссылки на объекты динамических переменных

Как отмечалось в разделе 11.1, ссылки *фактически* являются синонимами *объектов статических* переменных. Но объекты имеются и у *динамических* переменных. Поэтому направленные на них ссылки *также* не что иное, как синонимы динамических переменных (в общем случае объектов). Однако, если в использовании ссылок на простые типы данных (*int*, *float*, *double* и др.), практически нет никакого преимущества, то применение ссылок на динамические объекты удобно, например, тем, что в отличие от указателей их не требуется разыменовывать (не нужно ставить звёздочку перед именем ссылки). Как правило, это улучшает ясность программ.

Из очередного приложения *Ссылки на объекты* приводится только функция *PyskClick*. В её начале указатель *iDin* на тип *int* инициализируется динамическим объектом типа *int*. Далее, порождается ссылка *Ssul_int* на тип *int*, которая инициализируется объектом упомянутого указателя (в результате операции разыменования *iDin*). После порождения ссылки в упомянутый выше динамический объект (**iDin*) записывается число 2004, которое и выводится функцией *ShowMessage* при помощи обращения к ссылке *Ssul_int*. Вывод упомянутого числа оказался возможным благодаря тому, что ссылка *Ssul_int* смотрит на объект динамической переменной *iDin*. Согласно правилам хорошего стиля программирования далее динамический объект *iDin* корректно уничтожается.

```
void __fastcall TSsul_Objektu::PyskClick(TObject *Sender)
{
    int *iDin= new int;
    int &Ssul_int= *iDin;
    *iDin= 2004;
    ShowMessage(Ssul_int);//2004
    delete iDin;
    iDin= NULL;
    Label1->Caption= "Обращение по указателю";
    TLabel &Ssul_TLabel= *Label1;
    ShowMessage("Появится новое сообщение");
    Ssul_TLabel.Caption= "Обращение по ссылке";
} //PyskClick
```

Шесть первых строк тела функции можно заменить двумя:

```
int &Ssul_int= *new int(2004);
ShowMessage(Ssul_int);
```

В этом случае нет возможности уничтожить объект безымянного динамического объекта, на который смотрит ссылка *Ssul_int*.

В этом же приложении на форме с именем *Ssul_Objektu* имеется поле вывода с именем *Label1*. Вначале в нём программно (см. функцию *PyskClick*) выведено сообщение: *Обращение по указателю*. Ранее к свойствам визуальных динамических объектов (а более верно – указателей на визуальные динамические объекты) ин-

терфейса доступ осуществлялся при помощи «операция стрелка». Именно таким способом выводится упомянутое сообщение:


```
Label1->Caption= "Обращение по указателю";
```

Далее на тип *TLabel* объявляется ссылка *Ssul_TLabel*, она инициализируется объектом визуального объекта *Label1*. Теперь доступ к свойству *Caption* объекта *Label1* возможен при помощи «операция точка», применённой к ссылке *Ssul_TLabel*. Часто эту же операцию называют точечной нотацией. После осуществления задержки в работе программы с использованием вывода сообщения «Появится новое сообщение» прежний текст поля вывода замещается таким: «Обращение по ссылке». Это оказывается возможным в результате того, что объектом ссылки и указателя физически является один и тот же объект.

11.6. Осваиваем новые объекты интерфейса

Интерфейс должен быть удобным, интуитивно понятным и привлекательным для осуществления комфортного управления программой. Необходимо стремиться к тому, чтобы пользователи получали *интеллектуальное* наслаждение от работы с приложением. В этом разделе продолжим изучение элементов интерфейса, часто используемых в различных приложениях. Для компактности изложения их основных возможностей ниже рассматриваются упрощённые приложения с несколько надуманными и не всегда серьёзными задачами.

11.6.1. Переключатели

Переключатели – пожалуй, *обязательный* атрибут почти *любого* приложения. Они используются, например, при настройках различных параметров в офисных и других стандартных приложениях, без них не обходится выбор различных режимов работы самой операционной системы *Windows*, много их и в изучаемой среде быстрой разработки программ. Заготовка (шаблон) этого визуального объекта находится на вкладке *Standard* *Палитры Компонентов*, называется *RadioButton* (радиокнопка) и выглядит вот так . Применим этот объект для разработки приложения *Использование переключателей*, интерфейс которого показан на рис. 11.1.

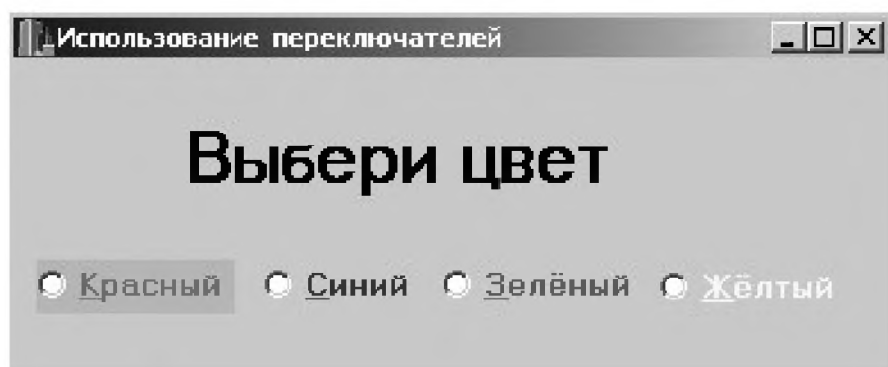


Рис. 11.1. Приложение *Использование переключателей*

При включении того или иного переключателя будет соответственно изменяться цвет сообщения, показанный в окне вывода *Label1*, а внутри выбранной радиокнопки устанавливается точка. Исходный цвет сообщения – чёрный. Все кнопки, расположенные на форме, объединяются в группу или логический блок. В группе может быть собрано две и более радиокнопок. Одиночные радиокнопки, как правило, не используются, вместо них удобнее применять индикаторы (они изучаются в следующем пункте).

Из кнопок, объединённых в группу, включённой (или нажатой, выбранной) может быть *лишь одна*: при щелчке по требуемой кнопке, кнопка, которая ранее была во включённом состоянии (выбрана), автоматически отключается (при этом точка исчезает). Именно в этом заключается назначение радиокнопок – обеспечить выбор *одного* из *нескольких* взаимоисключающих режимов.

Для реализации работы приложения *Использование переключателей* расположите на форме четыре экземпляра объектов типа *TRadioButton* и объект типа *TLabel* с именем *Label1*, в котором выведено сообщение *Выбери цвет* (см. рис. 11.1). При помощи свойства *Caption* каждой радиокнопки *справа* от них выводятся надписи – *красный, синий, зелёный, жёлтый* (соответствующими цветами, устанавливаемыми, например, в *Инспекторе Объектов* в свойстве *Font*). Для расположения надписи *слева* от кнопки в её свойстве *Alignment* устанавливается значение *taLeftJustify*. По умолчанию в этом свойстве выбрано значение *taRightJustify* (надпись справа), поскольку этот вариант удобен большинству пользователей.

Свойство *Checked* определяет, выбрана ли данная кнопка или нет. Оно позволяет *программно* осуществить назначение (и контроль) состояния радиокнопки. Перед началом работы приложения одна из кнопок обычно (но не всегда) включена (имеет внутри себя точку). С этой целью в *Инспекторе Объектов* в ходе разработки приложения её свойству *Checked* устанавливается значение *true*. Такую установку, конечно, можно осуществить и программно. Радиокнопки так устроены, что даже на этапе проектирования интерфейса значение *true* можно установить *лишь* для *одной* кнопки группы. В разрабатываемом приложении у всех кнопок в свойстве *Checked* установите значение *false*, сообщение выведите чёрным цветом и с использованием амперсанта подчеркните первую букву в названии цветов.

Для каждой кнопки породите заготовки функций, обрабатывающие событие *OnClick*. Для первой кнопки эта функция называться *RadioButton1Click*, в её тело впишите следующий код:


```
Label1->Font->Color= clRed;
```

Названия других функций будут отличаться лишь номером кнопки, а в их коде красный цвет (*clRed*) последовательно замените: синим (*clBlue*), зелёным (*clGreen*) и жёлтым (*clYellow*). Если требуется, чтобы после установки цвета выбранная кнопка исчезала с интерфейса и не появлялась при выборе других цветов, то ниже приведенной строки запишите код, который для первой кнопки выглядит так:

```
RadioButton1->Hide();
```

Метод *Hide()* (прятать) скрывает радиокнопку, делает её невидимой. Метод *Show()* позволяет вернуть радиокнопку на интерфейс: невидимую радиокнопку он делает вновь видимой. В упомянутых функциях применим оба метода, теперь, например, тело функции *RadioButton1Click* станет таким:

```
Label1->Font->Color= clRed;  
RadioButton1->Hide();  
ShowMessage("Ky-Ky");  
RadioButton1->Show();
```

Как отмечалось выше, все радиокнопки формы представляют собой *единую* группу, все они находятся *в одном* логическом блоке. Для порождения на форме *другой* группы радиокнопок их следует разместить в специальном контейнере. При помощи контейнеров на интерфейсе можно создать *любое* количество *независимых* между собой групп радиокнопок. Имеется целый ряд панелей, которые часто играют роль контейнеров для радиокнопок. Познакомимся с одним из таких объектов с именем *GroupBox*, его заготовка находится на вкладке *Standard Палитры Компонентов* и имеет следующий вид . Эту рамку называют *Групповой панелью*, *Контейнером с рамкой и надписью*. Контейнер на форме представляет собой рамку светло-серого цвета с заголовком (свойство *Caption*) в левом верхнем углу. К использованию контейнера прибегают также и из эстетических соображений, когда выбранные визуальные объекты интерфейса желают заключить в рамку. Это весьма целесообразно, если выбранная группа объектов предназначена для выполнения каких-то сходных или совместных задач.

Однако основное назначение контейнеров состоит в том, что бы все радиокнопки (или другие объекты), расположенные внутри него, заключить в *отдельном* логическом блоке, объединить в *отдельную* группу.

Для знакомства с объектом *GroupBox* изменим условие задачи предшествующего приложения. Пусть теперь прежняя группа радиокнопок изменяет цвет только первого слова сообщения (*Выбери*) по-прежнему записанного в свойстве *Caption* объекта *Label1*. Второе же слово сообщения (*цвет*) запишите в новом экземпляре объекта типа *TLabel* с именем *Label2*. Группу радиокнопок, предназначенную для изменения цвета слова *цвет*, расположите в объекте с именем *GroupBox1*. С этой целью *вначале* создайте объект *GroupBox1*, его габариты и местоположение на форме можно сделать такими же, как на рис. 11.2. Только после



Рис. 11.2. Приложение
Переключатели в контейнере





его порождения, внутри него, создавайте *новые* радиокнопки с соответствующими заголовками. Перетащить уже существующие радиокнопки (и другие объекты) внутрь контейнера *невозможно*.

В функциях по обработке нажатия радиокнопок этой группы впишите текст, аналогичный тексту для кнопок первой группы. Для первой кнопки второй группы (с именем *Button5*) он будет таким:

```
Label2->Font->Color= clRed;
GroupBox1->Font->Color= clRed;
```

После нажатия кнопки *Button5* изменится не только цвет текста в поле вывода *Label2*, но и заголовок объекта *GroupBox1*. В функциях для 6, 7 и 8 кнопок следует лишь красный цвет заменить соответственно: синим, зелёным и жёлтым.

11.6.2. Индикаторы

Индикатор с флажком или просто индикатор *CheckBox*  (вкладка *Standard*) используются в приложениях для включения, выключения и индикации *определённых* режимов (или опций). По умолчанию в его свойстве *AllowGrayed* (установка дополнительного состояния) выбрано значение *false*. В этом случае свойство *State* (состояние) имеет лишь два положения: *cbChecked*  (выбрано, *cb* – от *CheckBox*) и *cbUnchecked*  (не выбрано). Установка в свойстве *AllowGrayed* значения *true* позволяет получить *дополнительное* (третье) состояние свойства. Оно называется *cbGrayed* (серое) и имеет вид:  (птичка серого цвета).

Если на интерфейсе приложения (в ходе работы программы) в индикаторе, например, *CheckBox1* поставлена галочка (щелчком мыши), то при *AllowGrayed == false* это эквивалентно таким назначениям:

```
CheckBox1->Checked= true;
CheckBox1->State= cbChecked;
```

При выполнении в программе *одной* из строк приведенного кода в объекте *CheckBox1* будет установлена галочка. Если программно либо в *Инспекторе Объектов* свойству *Checked* назначить *false*, то свойство *State* получит значение *cbUnchecked* и в объекте *CheckBox1* галочка исчезнет. Таким образом, упомянутые свойства связаны между собой и предназначены в основном для выполнения одних и тех же действий. Следует отметить, что при *любом* значении (*true* или *false*) свойства *AllowGrayed* свойству *State* можно назначить *любое* значение (*cbChecked*, *cbUnchecked* или *cbGrayed*).

Индикаторы можно использовать вместо переключателей. Проиллюстрируем эту возможность в приложении *Использование индикаторов*, интерфейс которого показан на рис. 11.3. В этом приложении каждый индикатор имеет только *два* положения. При щелчке мышкой по любому из них изменяется цвет надписи, в самом индикаторе появляется птичка (галочка), при повторном щелчке по этому же индикатору галочка исчезает, однако цвет текста остаётся прежним. При щелчке по *любому* другому индикатору птичка из ранее включенного индикатора

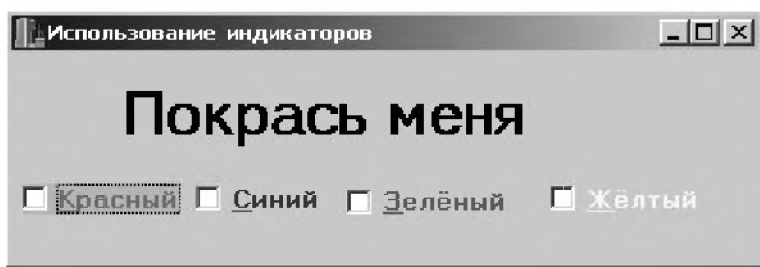


Рис. 11.3. Приложение
Использование индикаторов

исчезнет, а надпись окрасится новым, выбранным вами цветом. Файл реализации приложения показан ниже.

//Файл реализации приложения *Использование индикаторов*.

```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
```

```
{
    if (CheckBox1->Checked== true)
        Label1->Font->Color= clRed;
    CheckBox2->Checked= false;
    CheckBox3->Checked= false;
    CheckBox4->Checked= false;
} //CheckBox1Click
```

```
void __fastcall TForm1::CheckBox2Click(TObject *Sender)
```

```
{
    if (CheckBox2->Checked== true)
        Label1->Font->Color= clBlue;
    CheckBox1->Checked= false;
    CheckBox3->Checked= false;
    CheckBox4->Checked= false;
} //CheckBox2Click
```

```
void __fastcall TForm1::CheckBox3Click(TObject *Sender)
```

```
{
    if (CheckBox3->Checked== true)
        Label1->Font->Color= clGreen;
    CheckBox1->Checked= false;
    CheckBox2->Checked= false;
    CheckBox4->Checked= false;
} //CheckBox3Click
```

```
void __fastcall TForm1::CheckBox4Click(TObject *Sender)
```

```
{
    if (CheckBox4->Checked== true)
        Label1->Font->Color= clYellow;
    CheckBox1->Checked= false;
    CheckBox2->Checked= false;
    CheckBox3->Checked= false;
} //CheckBox4Click
```

Покажем теперь использование *третьего* состояния индикатора на примере приложения *Индикаторы* с интерфейсом, приведенным на рис. 11.4. В заголовках трёх кнопок типа *TButton* (с именами *X_Bolshe*, *X_Menjshe* и *X_Pavno*) указаны три условия. При нажатии любой из кнопок (выборе соответствующего условия) под кнопкой появляется индикатор, указывающий, какое условие выбрано в текущий момент. При нажатии первой, второй и третьей кнопок соответственно

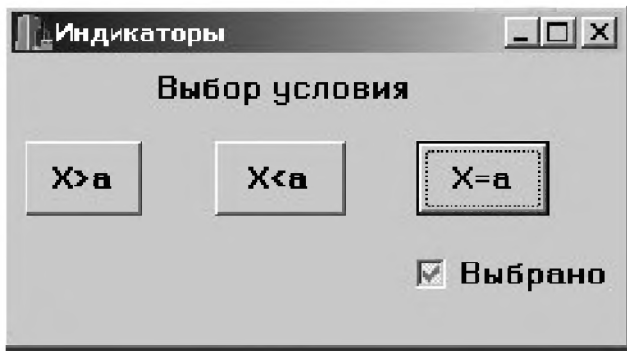


Рис. 11.4. Приложение Индикаторы

появляются окошко с галочкой, окошко без галочки и окошко с серой галочкой. Показанный интерфейс соответствует случаю, когда пользователь нажал третью командную кнопку, что и обеспечило вывод серого варианта индикатора. Файл реализации показан ниже.

```
// файл реализации приложения Индикаторы.
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    //При запуске программы индикатор не виден
    CheckBox1->Visible= false;
}

void __fastcall TForm1::X_BolsheClick(TObject *Sender)
{
    CheckBox1->Visible= true; //Индикатор становится видимым
    CheckBox1->State= cbChecked;
    CheckBox1->Left= X_Bolshe->Left; //Смещение индикатора
}

void __fastcall TForm1::X_MenjsheClick(TObject *Sender)
{
    CheckBox1->Visible= true;
    CheckBox1->State= cbUnchecked;
    CheckBox1->Left= X_Menjshe->Left;
}

void __fastcall TForm1::X_PavnoClick(TObject *Sender)
{
    CheckBox1->Visible= true;
    CheckBox1->State= cbGrayed;
    CheckBox1->Left= X_Pavno->Left;
}
```

В приложении для смещения индикатора *CheckBox1* в горизонтальном направлении используется его свойство *Left*. Это свойство имеется у *всех* визуальных объектов интерфейса, в нём устанавливается расстояние в пикселях *левой* границы объекта от *левого* края интерфейса. Местоположение индикатора изменяется в результате присваивания его свойству *Left* значения свойства *Left* выбранной кнопки (*X_Bolshe*, *X_Menjshe* или *X_Pavno*). В свойстве *Top* визуальных объектов устанавливается расстояние между верхней кромкой объекта и верхней границей интерфейса, его можно использовать как для программной установки местоположения, так и смещения объекта по вертикали.

В рассматриваемом упражнении для скрытия объекта *CheckBox1* использовано свойство *Visible* со значением *false*, а для его показа – со значением *true*. Однако для этих же целей применимы соответственно методы *Hide()* и *Show()*, использование которых иллюстрировалось в п. 11.6.1 для объекта типа *TRadioButton*. Вместе с тем, у объектов типа *TRadioButton* также имеется свойство *Visible*. Как видите, одни и те же действия можно реализовать как применением свойств визуальных объектов, так и их методов.

11.6.3. Диалоговое окно

У пользователей в ходе их диалога с приложениями ведущих фирм обычно имеется выбор из двух или более альтернатив. Однако до настоящего времени в наших учебных приложениях пользователь для продолжения работы приложения вынужден нажимать *только* кнопку *Да* в окне функции *ShowMessage*. Вместе с тем в большинстве приложений *Windows* и в ходе работы самой этой операционной системы (они, безусловно, являются образцами для подражания), пользователя, например, предупреждают о том, что введенная им команда может быть исполнена, отменена, либо по ней можно получить справочную информацию. Бывают и другие варианты диалогов (с одним, двумя и большим числом альтернатив).

Такие диалоги являются общепринятым стандартом, ведь люди склонны изменять свои намерения, могут ввести ту или иную команду по ошибке, поэтому требуется её отмена. В творческой деятельности без «откатов» вперед-назад просто невозможно обойтись. Не будем далее *агитировать вас за советскую власть*, а научим применять один из самых удачных диалогов *Builder*.

Из имеющихся в *Builder* *нескольких* функций, порождающих окна диалогов, только функция *MessageBox* позволяет в русифицированной *Windows* *все* сообщения выводить на русском языке. Вместе с тем эта же функция для англоязычного варианта указанной операционной системы диалог ведёт по-английски.

У этой функции имеется *три* формальных параметра. Первые два из них – текстового типа (тип *PChar*), и предназначены соответственно для вывода сообщения для пользователя и заголовка диалогового окна. Число символов в этих параметрах может превышать 255. Длинные *сообщения* окна вывода *автоматически* переносятся на другую строку, а длинный текст заголовка окна диалога прерывается троеточием (однако всплывающая подсказка показывает весь текст).

Третий параметр, так называемый *флаг*, совместим по типу с типом *int* (его тип *Longint*) и предназначен для выбора конкретной разновидности диалогового окна (они отличаются количеством альтернатив и их заголовками (*Да*, *Нет*, *Отмена* и т. д.)) и поясняющего значка (например, внимание, вопрос и др.). Возвращает эта функция целочисленное значение, совместимое с типом *int* (тип *Integer*). Здесь типы параметров, указанные в скобках, относятся к *Object Pascal*: этот язык программирования использован для разработки *Builder*.

Применение функции *MessageBox* иллюстрирует приложение *Изучаем*

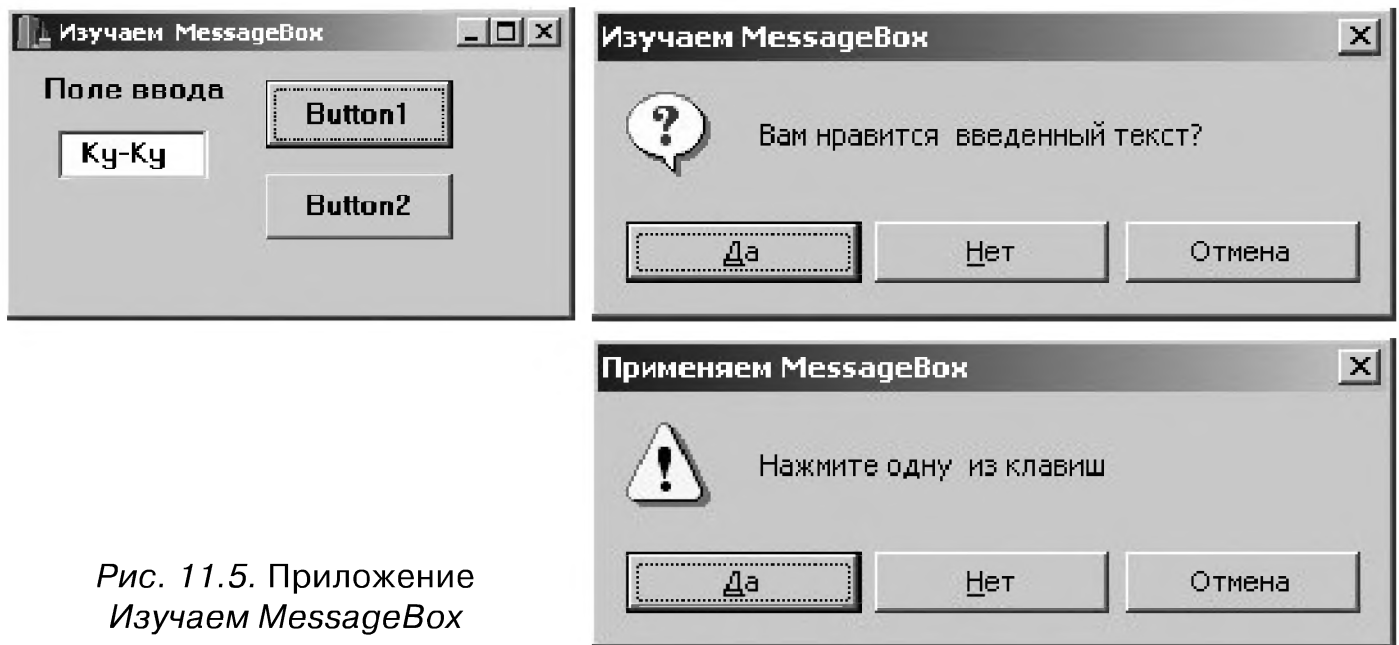


Рис. 11.5. Приложение
Изучаем MessageBox

MessageBox. На рис. 11.5 показаны интерфейс этого приложения (левая панель) и диалоговые окна, всплывающие после нажатия кнопок *Button1* (правая верхняя панель) и *Button2* (правая нижняя панель).

Нажатие кнопок *Button1* и *Button2* обрабатывают соответственно функции *Button1Click* и *Button2Click*. В первой из этих функций используются строковые константы для выбора вида диалога, а во второй функции – их числовые эквиваленты.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int Znach= Application->MessageBox("Вам нравится\
        введенный текст?", "Изучаем MessageBox",
        MB_YESNOCANCEL + MB_ICONQUESTION);

    switch (Znach)
    {
        case IDCANCEL: ShowMessage("Выходим из приложения");//2
                        Close();
        case IDYES: ShowMessage("Продолжайте работу");//6
                   break;
        case IDNO: ShowMessage("Введите новый текст");//7
    } //switch
} //Button1Click

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    int Znach= Application->MessageBox("Нажмите одну\
        из клавиш", "Применяем MessageBox",
        MB_YESNOCANCEL + MB_ICONWARNING);

    switch (Znach)
    {
        case 2: ShowMessage("Выбрана клавиша \"Отмена\" (Cancel)");
                break;
        case 6: ShowMessage("Выбрана клавиша \"ДА\" (Yes)");
                break;
        case 7: ShowMessage("Выбрана клавиша \"Нет\" (No)");
    } //switch
} //Button2Click
```


Функция *MessageBox* является методом класса *Application*, доступ к ней осуществляется через операцию стрелка, упомянутый класс подключается к каждому приложению автоматически. Первые два текстовых параметра этой функции используются соответственно для вывода сообщения и заголовка окна (см. правые панели рис.11.5).

Третий параметр-флаг функции *MessageBox* может состоять из двух частей. Первая часть является обязательной, она предназначена для определения *вида* диалогового окна. Ниже рассмотрены *все* разновидности таких окон. Сейчас же сделаем расшифровку лишь применённого типа *MB_YESNOCANCEL*. Здесь используется выделение фрагментов идентификатора для большей ясности чтения слов в имени применённого диалогового окна, включающего три кнопки с надписями *Да*, *Нет* и *Отмена* (см. рис.11.5). Начальные две буквы *MB* – происходят от имени функции *MessageBox*. Числовые варианты использованных типов диалоговых окон показаны в комментариях в функции *Button1Click*.

Вторая, необязательная, часть флага – это иконка. Её вывод осуществляется либо с использованием знака «+» (например, *MB_YESNOCANCEL + MB_ICONQUESTION*), либо вертикальной черты (побитового ИЛИ) (*MB_YESNOCANCEL | MB_ICONQUESTION*). Как можно видеть на панелях, расположенных на рис.11.5 справа, применённые иконки содержат знак вопроса (функция *Button1Click*) и восклицательный знак (функция *Button2Click*).

Функция *MessageBox* возвращает целочисленные значения (числа от 0 до 7), однако, все они, кроме нуля, дублируются также текстовыми константами, наглядно подсказывающими назначение нажатых (выбранных) кнопок. В нашем примере это *IDCANCEL*, *IDYES* и *IDNO*. Начальные две буквы *ID* происходят от слов *inform dialog* (информационный диалог или вопросы диалога).

Все виды диалогов функции *MessageBox* приведены в следующей таблице.

Наименование флагов	Надписи на кнопках		
<i>MB_ABORTRETRYIGNORE</i>	Стоп (<i>Abort</i>)	Повтор (<i>Retry</i>)	Пропустить (<i>Ignore</i>)
<i>MB_OK</i>	ОК		
<i>MB_OKCANCEL</i>	ОК	Отмена (<i>Cancel</i>)	
<i>MB_RENRYCANCEL</i>	Повтор (<i>Retry</i>)	Отмена (<i>Cancel</i>)	
<i>MB_YESNO</i>	Да (<i>Yes</i>)	Нет (<i>No</i>)	
<i>MB_YESCANCEL</i>	Да (<i>Yes</i>)	Нет (<i>No</i>)	Отмена (<i>Cancel</i>)





Имеется ещё два дополнительных флага. Флаг *MB_TOPMOST* помещает окно выводимого диалога всегда поверх других окон, даже в случае, когда открыто другое приложение. Пользователя можно проинформировать справочной информацией по введенной команде или наступившего события, если прибегнуть к дополнительному флагу *MB_HELP*. При использовании этого флага в диалоговом окне появится дополнительная клавиша с заголовком *Справка (Help)*, щелчок по которой генерирует событие *Help* и эквивалентен нажатию функциональной клавиши *F1*. Напоминаем, выводятся дополнительные флаги с

использованием знака «+» или «|».

Если памяти недостаточно для вывода диалогового окна, то функция *MessageBox* возвращает нуль. При успешном исполнении этой функции возвращаемые значения зависят от имени нажатой кнопки:

Значение	Числовой эквивалент	Нажатая кнопка
<i>IDOK</i>	1	ОК
<i>IDCANCEL</i>	2	Отмена, <i>Esc</i>
<i>IDABORT</i>	3	Стоп
<i>IDRETRY</i>	4	Повтор
<i>IDIGNORE</i>	5	Пропустить
<i>IDYES</i>	6	Да
<i>IDNO</i>	7	Нет

Флаги пиктограмм и изображения, которые они выводят, показаны в таблице:

Флаг	Пиктограмма
<i>MB_ICONEXCLAMATION</i> , <i>MB_ICONWARNING</i>	
<i>MB_ICONINFORMATION</i> <i>MB_ICONASTERISK</i> (звёздочка)	
<i>MB_ICONQUESTION</i>	
<i>MB_ICONSTOP</i> , <i>MB_ICONERROR</i> , <i>MB_ICONHAND</i>	

Существует ещё два флага, они называются флагами модальности. Один из них *MB_APPLMODAL* – принят по умолчанию. В случае его применения пользователь *обязательно* должен нажать одну из клавиш, если желает продолжить дальнейшую работу приложения. При этом *имеется* возможность перейти в окна *других приложений* или всплывающие окна этого же приложения.

Флаг *MB_SYSTEMMODAL* используется для предупреждения пользователя о серьёзных ошибках, которые требуют *немедленного* принятия мер. Поэтому это окно *всегда* остаётся *поверх* других окон, даже в случае перехода в иные приложения. Другие свойства совпадают с флагом *MB_APPLMODAL*.

11.6.4. Диаграммы длительных процессов

В различных приложениях *Windows* пользователя *всегда* информируют о начале и ходе длительного процесса. Так как такой процесс может продолжаться от нескольких секунд до нескольких десятков минут и более, то без специального предупреждения пользователь может ошибочно предположить, что программа заиклилась (зависла) и поэтому станет предпринимать какие-то действия. Чтобы этого не случилось, выводят *диаграммы длительных процессов*. Они применя-

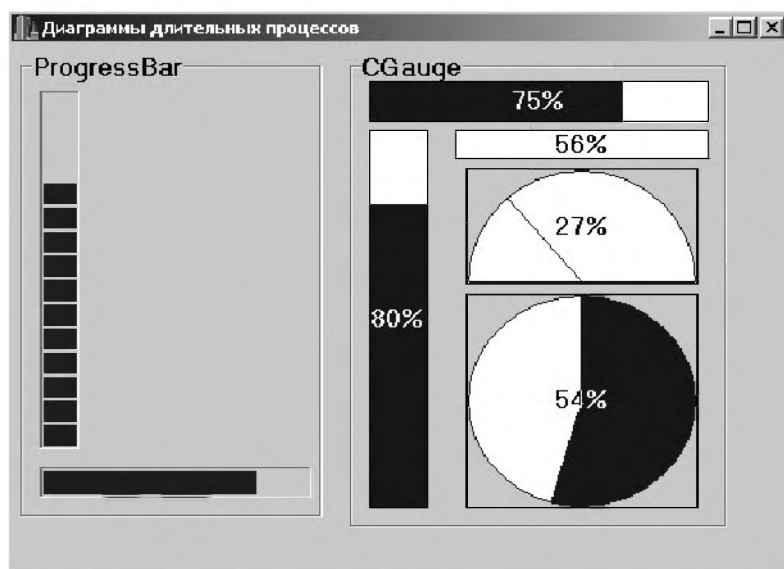



Рис. 11.6. Примеры разновидностей диаграмм

ются, например, при копировании файлов в *FAR* и в работе программы *Проверка диска*. В этом пункте рассматриваются два вида диаграмм длительных процессов: *ProgressBar* (растущая полоса, вкладка *Палитры Компонентов Win32*)  и *CGauge* (оценка времени, вкладка *Samples*) . Разновидности этих диаграмм показаны на рис 11.6.

Применим эти диаграммы в приложении-шутке с именем *Сюрприз*, в котором пользователю предлагается отформатировать жёсткий диск *C*. При согласии и несогласии пользователя на такую неожиданную *услугу* программа информирует его о том, что для получения *максимального* удовольствия от сюрприза диск требуется отформатировать, и что процесс форматирования *запущен*. При этом выводится диаграмма *ложного* процесса форматирования, по завершению которого осуществляется перезагрузка (или выключение) компьютера.

Для такой перезагрузки используем функцию из библиотеки *Windows API* (*Application Programming Interface* – программирование интерфейса приложений) с именем *ExitWindowsEx* (выход из *Windows*). У этой функции имеется два параметра. В качестве фактического параметра первого аргумента используем режим перезагрузки *EWX_REBOOT*, а второго – произвольную переменную или константу типа *DWORD* (его значения находятся в интервале от 0 до 4 294 967 295). Последний параметр является резервным, он *может* использоваться в последующих модернизированных вариантах этой функции. Если требуется выключить компьютер, то в функции *ExitWindowsEx* вместо первого параметра следует записать: *EWX_SHUTDOWN || EWX_POWEROF*.

В приложении *Сюрприз* для вывода текста применим компонент *StaticText* (метка с бордюром) . Он находится на вкладке *Additional Палитры Компонентов*, по своему назначению подобен компоненту *Label*, но обеспечивает возможность задания *стиля* бордюра (рамки). Кроме того, если установить его свойству *AutoSize* значение *false*, а его ширину сделать *меньше* ширины выводимого текста, то произойдёт *автоматический* перенос оставшейся части текста (превышающей ширину окна вывода) на *следующую* строку. При этом высоту упомянутого

объекта необходимо растянуть на нужное количество выводимых строк в *Инспекторе объектов* либо в *Визуальном проектировщике*.

Вначале познакомимся с объектом-диаграммой типа *TProgressBar*. В разрабатываемом приложении оставьте ему имя по умолчанию *ProgressBar1*. Далее, растяните этот объект *по горизонтали* в нижней части формы с именем *Form1*. У этого объекта имеются свойства *Min* и *Max*, предназначенные для записи в них (программно или в *Инспекторе Объектов*) минимальных и максимальных значений свойства *Position*, отображающего текущее положение индикатора. По умолчанию в свойствах *Min* и *Max* находятся числа соответственно 0 и 100: это минимальное и максимальное значения свойства *Position*, выраженные в процентах относительно максимального значения этого свойства.

В приложении *Сюрприз* используем ещё одну функцию из библиотеки *Windows API*: функцию *GetTickCount()*, которая возвращает время в миллисекундах, прошедшее после запуска *Windows*. Тип возвращаемого ею значения – *DWORD*. С использованием этой функции определим время начала и конца процесса, в течение которого будет *мучиться* пользователь приложения *Сюрприз* (длительность этого процесса зададим константой *T*). Время начала и конца процесса будем хранить в переменных типа *DWORD* с именами соответственно *Nachalo* и *Konec*.

```
const DWORD T= 10000; //соответствует 10 секундам
DWORD Nachalo= GetTickCount(), //Начало процесса в миллисек.
Konec= Nachalo + T; //Момент его завершения
```

Передвижение полосы индикатора в течение времени *T* осуществляется следующим фрагментом кода:

```
while (GetTickCount() < Konec)
    ProgressBar1->Position= 100*(GetTickCount() - Nachalo)/T;
```

На каждом шаге приведенного цикла *while* в свойство *Position* записывается *текущее* относительное значение пройденной части заданного интервала *T*, выраженное в процентах.

Можно эти же действия реализовать и таким кодом:

```
ProgressBar1->Max= T;
while (GetTickCount() < Konec)
    ProgressBar1->Position= GetTickCount() - Nachalo;
```

В этом случае *Builder* сам выразит в процентах разность *GetTickCount() - Nachalo* относительно величины *T= 10000*. Если в *Инспекторе Объектов* в свойстве *Max* записать значение 10000 (вместо исходного 100), то в предшествующем коде можно убрать первую строку.

Увидев грозное предупреждение о *неминуемом* форматировании жёсткого диска, пользователь попытается нажать имеющуюся на интерфейсе командную кнопку с заголовком *Выход* (её имя *Vuxod*). Поэтому заблокируем функционирование этой кнопки следующим кодом:

```
Vuxod->Enabled= false; //Блокируется кнопка Выход
```

При этом сделаем так, чтобы *все* граничные пиктограммы интерфейса (в том числе и крестик) также оказались заблокированными. Как это осуществляется,

расскажем несколько позже.

Пользователь программы может попытаться *снять* выполнение такого *сюприза* при помощи ввода известной команды *Ctrl+Alt+Del*. Поэтому замаскируем в выведенном списке запущенных программ программу *Сюрприз* таким способом: её имя заменим пробелом, либо именем стандартной программы, которая вероятнее всего может иметься на компьютере заказчика-жертвы вашей шутки.

```
Form1->Caption= " "; //или "Explorer", "Word";
```

Если вместо пробела поставить пустую строку, то в этом случае *Windows* возьмёт имя исполняемого файла программы *Сюрприз*, что позволит пользователю легко «вычислить» диверсанта и прекратить его работу (снять программу с выполнения).

После завершения ложного форматирования (работы объекта *ProgressBar1*) восстановим функционирование граничных пиктограмм и кнопки *Выход* (уместно принести ещё и извинения за доставленные беспокойства). Напоминаем, что в *среде программирования* прерывание работы программы осуществляется командой *Ctrl+F2 (Run/Program Reset)*. Ниже приводятся пользовательские функции файла реализации.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    if (Application->MessageBox("Для получения наибольшего\
удовольствия от сюрприза необходимо отформатировать диск C:.\
Продолжить?", "Ваш выбор", MB_YESNO|MB_ICONQUESTION)==IDYES)
        StaticText1->Caption= "Хорошо. \nНачинаем форматирование\
        диска C:\nЭто может занять несколько минут...";
    else
        StaticText1->Caption= "Программа решила предоставить Вам\
        максимальное удовольствие от сюрприза.\nНачинаем\
        форматирование диска C:\nЭто может занять несколько\
        минут...";
    Vuxod->Enabled= false; //Блокируется кнопка Выход
    Form1->Caption= " "; //«Explorer», "Word" — маскировка
    const DWORD T= 10000; //10 с
    DWORD Nachalo= GetTickCount(),
        Konec= Nachalo + T;
    while (GetTickCount() < Konec)
        ProgressBar1->Position= 100*(GetTickCount() - Nachalo)/T;
    Vuxod->Enabled= true;
    Form1->Caption= "Сюрприз"; //Восстанавливаем прежний заголовок приложения
    StaticText1->Caption= "Форматирование диска C: завершено";
    Application->MessageBox("Необходима перезагрузка. \nВыполнить её сейчас?",
        "Перезагрузка", MB_YESNO | MB_ICONQUESTION);
    StaticText1->Caption= "Системе очень нужна перезагрузка. \nПроцесс пошёл...";
    ExitWindowsEx(EWX_REBOOT, Nachalo);
    // или ExitWindowsEx(EWX_SHUTDOWN || EWX_POWEROF, 0);
} // Button1Click

void __fastcall TForm1::VuxodClick(TObject *Sender)
{
    Close();
} // VuxodClick
```

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    Action= caNone;
    if (Vuxod->Enabled== true)
        Action= caHide;//или caFree; caMinimize
} // FormClose
```

После щелчка по крайней справа (крестику) граничной пиктограмме интерфейса функция *FormClose* делает не активными (блокирует) *все* граничные пиктограммы и возобновляет (разблокирует) их функционирование *после* снятия блокировки с кнопки *Выход*. Заготовка этой функции появляется (в файле реализации) с использованием двойного щелчка в поле ввода события формы *OnClose*. Ссылка на переменную *Action* в заголовке функции *FormClose* обеспечивает отслеживание *всех* действий, выполняемых в программе по её закрытию.

При нажатии граничной пиктограммы *крестик*, переменной *Action* назначается значение *caNone* (*None* – ничего), в результате осуществляется блокировка выхода из приложения. В константах-перечислениях *caNone*, *caHide*, *caFree*, *caMinimize* первые две буквы происходят от слов *Close Action*. Когда значение свойства *Enabled* кнопки *Vuxod* изменится с *false* на *true* (кнопка разблокируется), произойдёт и разблокировка граничных пиктограмм интерфейса: переменной *Action* присваивается значение *caHide*. Эта переменная может принимать ещё и значения *caFree*, *caMinimize*, использование которых в данной программе приводит к тем же результатам, что и в случае значения *caHide*.

Выше использована диаграмма *ProgressBar*, однако аналогичные действия реализуются и при помощи диаграммы *CGauge*. Для её испытания в предшествующей программе вместо фрагмента

```
ProgressBar1->Position= 100*(GetTickCount() - Nachalo)/T;
```

запишите такой код:

```
CGauge1->Progress= 100*(GetTickCount() - Nachalo)/T;
```

Конечно, при этом необходимо расположить объект *CGauge1* на интерфейсе. В упомянутом объекте положение перемещающейся правой границы полосы определяется свойством *Progress* (текущее положение). Свойства упомянутых диаграмм очень близки между собой, поэтому их удобно описать в единой таблице.

Свойства <i>ProgressBar</i>	Свойства <i>CGauge</i>	Описание
<i>Max</i>	<i>MaxValue</i>	Максимальное значение свойства <i>Position</i> или <i>Progress</i> , при котором отображаемый процесс завершается. По умолчанию задаётся 100%
<i>Min</i>	<i>MinValue</i>	Исходное значение свойства <i>Position</i> или <i>Progress</i> , оно соответствует началу отображаемого процесса.
<i>Position</i>	<i>Progress</i>	Текущее положение индикатора между его минимальным и максимальным значениями, характеризует часть выполненного процесса, выраженную в процентах.

Свойства ProgressBar	Свойства CGauge	Описание
<i>Smooth</i>	–	Непрерывное (при <i>true</i>) или дискретное (при <i>false</i>) отображение процесса.
<i>Step</i>	–	Шаг приращения позиции индикатора, используемый в методе <i>StepIt</i> . По умолчанию <i>Step=10</i> .
<i>Orientation</i>	–	<i>pbHorizontal</i> , <i>pbVertical</i> . Объект необходимо вытягивать в соответствующем направлении.
–	<i>ForeColor</i>	Цвет заполнения.
–	<i>ShowText</i>	Показ на фоне диаграммы числа – текущей доли выполненного процесса, выраженной в процентах.
–	<i>Kind</i>	Определяет тип диаграммы: <i>gkHorizontalBar</i> , <i>gkVericalBar</i> , <i>gkPie</i> – круговая, <i>gkNeedle</i> – секторная, <i>gkText</i> – только числа.

У объектов типа *TProgressBar* имеются два метода. Метод *StepIt* увеличивает позицию указателя диаграммы на один шаг, величина которого задаётся свойством *Step*. Для иллюстрации метода *StepIt* разработаем приложение, на интерфейсе которого разместим только поле ввода *Edit1*, а в тело функции *Edit1Change* (обработка события *OnChange*) запишем две строки:

```
ProgressBar1->Step= StrToInt(Edit1->Text);
ProgressBar1->StepIt();
```

В упомянутом приложении при каждом новом вводе значений в поле объекта *Edit1* на диаграмме *ProgressBar1* будет происходить смещение полосы индикатора на соответствующее число процентов.

Метод *StepBy(int Shag)* позволяет эти же действия выполнить одной строкой

```
ProgressBar1->StepBy(StrToInt(Edit1->Text));
```

Этот метод увеличивает позицию индикатора на величину *Shag*.

11.6.5. Непослушная кнопка

Положение объектов, расположенных на форме, определяется координатами их верхнего левого угла. Эта точка интерфейса задаётся расстояниями (выраженными в пикселях) от верхней (*top*) и левой (*left*) границ формы. Указанные удаления верхнего левого угла перемещаемого объекта *автоматически* записываются в его свойства соответственно *Top* и *Left*. Ясно, что упомянутые значения можно задавать и программно (подробнее см. п. 11.6.2, приложение *Индикаторы*). Ниже приведена функция *Button1MouseMove*, её заготовка порождается при помощи события *OnMouseMove* для командной кнопки *Button1*. При приближении указателя мыши к границам кнопки её местоположение изменяется по случайному закону, определяемому функцией *random*. Поэтому пользователь этого приложения-шутки *никогда* не может осуществить призыв, начертанный в заголовке кнопки: *Нажми меня*.

С тем, чтобы кнопка не переместилась за границы формы, её свойствам *Top* и

Left назначаются случайные значения в пределах соответствующих габаритов формы, уменьшенных на половину соответственно высоты и ширины кнопки.

```
void __fastcall TForm1::Button1MouseMove(TObject *Sender,  
                                         TShiftState Shift, int X, int Y)  
{  
    Button1->Left= random(Form1->Width - Button1->Width/2);  
    Button1->Top= random(Form1->Height - Button1->Height/2);  
} //Button1MouseMove
```

11.6.6. Модальные и немодальные формы

Основные свойства

В больших приложениях (например, *Word*, *Excel*, *Builder* и др.) для осуществления выбора и установки различных параметров (настроек) прибегают к использованию *дополнительных* интерфейсов (форм), что существенно упрощает управление программой. Поэтому далее расскажем о различных типах форм и методах их включения в приложения.

Прежде всего, отметим, что в *каждом* приложении *обязательно* существует *главная форма*, и могут иметься подчинённые ей *модальные* и *немодальные* формы. При наличии только одной формы она имеет статус главной (является главной формой). При порождении нескольких форм по умолчанию *первая* из них назначается главной, а остальные – *вспомогательными* формами. Однако позже *любую* вспомогательную форму легко сделать главной (и наоборот). Кроме того, можно управлять *видимостью форм*: по ходу работы приложения делать их видимыми или невидимыми.

Перечислим *основные свойства* главной формы:

- ☐ Главной форме передаётся управление *в начале* выполнения приложения.
- ☐ При закрытии главной формы работа приложения *завершается*.
- ☐ Главную форму можно сделать невидимой только в случае, когда *не закрыта* хотя бы одна из вспомогательных форм приложения. При отсутствии подчинённых форм главная форма всегда видима на экране.

Модальная форма после завершения всех её установок должна *закрываться*. При попытке перейти в другую форму услышите короткий звуковой сигнал, свидетельствующий о том, что работа программы приостановлена *до закрытия* модальной формы.

Немодальная форма *может не закрываться*: это не препятствует дальнейшей работе приложения. Такой режим подчинённой формы также часто востребован при разработке приложений.

Пример использования трёх форм

Как использовать *несколько* форм, разъясним на примере приложения *Модальные и немодальные формы*, все интерфейсы которого показаны на рис.11.7. При запуске этого приложения вначале появится главная форма-приветствие

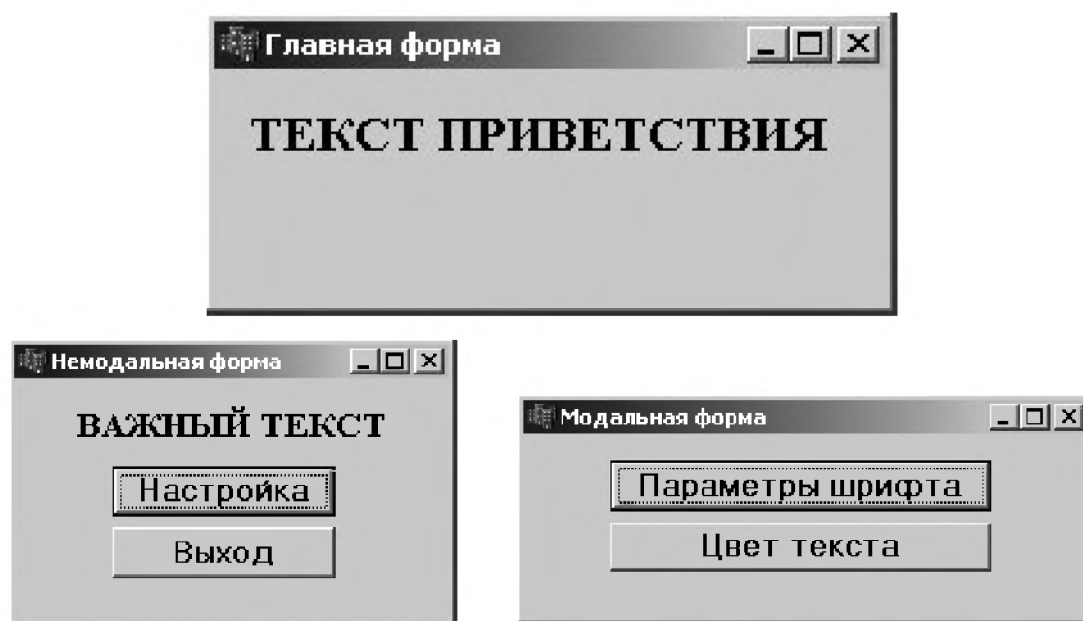


Рис. 11.7. Приложение Модальные и немодальные формы

(верхняя панель на рис. 11.7), которая через 5 секунд сменится немодальной формой (левая нижняя панель на рис. 11.7) с сообщением *Важный текст* (в окне объекта *Label1*), двумя командными кнопками с заголовками *Настройка* (имя *Nastrojka*) и *Выход* (имя *Vuxod*). Выход из этого приложения осуществляется *только* при помощи кнопки *Выход* (поскольку главная форма через 5 секунд после начала работы приложения окажется невидимой, поэтому недоступной). При нажатии кнопки *Настройка* открывается модальная форма (правая нижняя панель на рис. 11.7), на которой имеется только две командные кнопки с заголовками *Параметры шрифта* (имя *Shrift*) и *Цвет текста* (имя *Cvet*). При нажатии кнопки *Цвет текста* появляется стандартное окно *Windows* для выбора цвета текста сообщения *Важный текст*, расположенного на немодальной форме. Кнопка *Параметры шрифта* выводит стандартное окно для выбора параметров текста (имя, начертание, размер и цвет шрифта).

Приступим теперь к разработке упомянутых форм и их файлов реализации. Форма, заготовка которой появляется автоматически при открытии нового приложения, пусть будет главной (с именем *Glav_Form*). На этой форме расположите объект *Label1* и невидимый объект *Timer1* (вкладка *System*), предназначенный для задания интервала времени (в свойстве *Interval*), по истечению которого (измеряется в миллисекундах) наступает событие *OnTimer*. Упомянутый интервал можно задать в *Инспекторе Объектов* (по умолчанию там установлено 1000 (1 с)), однако в настоящем приложении зададим его в функции *FormCreate* (событие *OnCreate*), где также выведем текст приветствия (эта форма может играть роль логотипа приложения):

```
Label1->Caption= "ТЕКСТ ПРИВЕТСТВИЯ";  
Timer1->Interval= 5000;//5 с
```

В обработчике события *OnTimer* (оно наступает по истечению 5 с *после* запуска приложения) запишем операции по скрытию главной формы, порождения не-

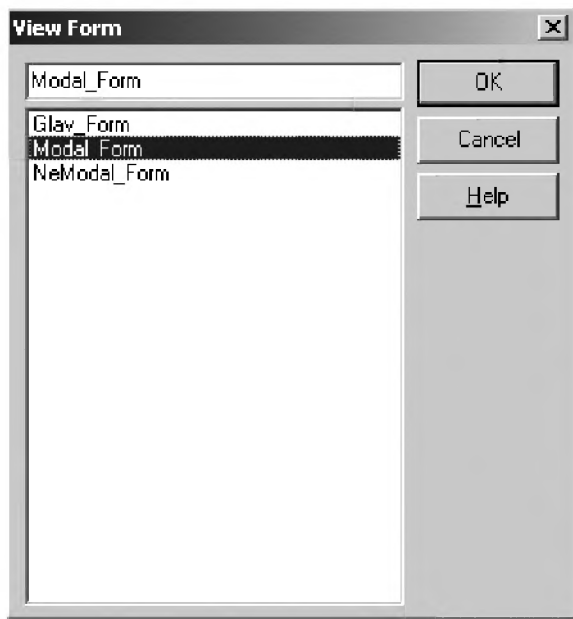


Рис. 11.8. Выбор одной из форм


модальной формы и отключение работы таймера:


```
Glav_Form->Visible= false; // Скрываем Glav_Form
NeModal_Form->Show(); //Показываем NeModal_Form
Timer1->Enabled= false; // Отключение Timer1
```


Метод *Show()* делает форму немодальной и показывает (делает видимой) её на экране. Для отключения таймера вместо использования свойства *Enabled* (доступность) можно свойству *Interval* назначить нулевое значение:


```
Timer1->Interval= 0; //Отключение таймера
```

На наш взгляд, предшествующий вариант кода является более ясным.

Порождение *вспомогательных* форм осуществляется вводом команды горизонтального меню *File/New Form*, либо щёлчком по пиктограмме на *Панели Инструментов*  (она называется *New Form*). Интерфейс каждой из вспомогательных форм оформите так, как показано на нижней части рис. 11.7, приведенным на них объектам назначьте упомянутые выше имена, а формы назовите соответственно *NeModal_Form* и *Modal_Form*.

Для вывода одной из ранее порождённых форм можно применить команду *View/Forms, Shift+F12* либо иконку  (*View Form*). В любом из этих вариантов команды появится окно выбора форм (см. рис. 11.8).

У каждой формы (главной и вспомогательных) имеется *свой* файл реализации (с расширением *.cpp*), переход между формой и этим файлом (и обратно) осуществляется функциональной клавишей *F12*. Этот же переход можно совершить и кнопкой на *Панели Инструментов*  (*Toggle Form/Unit* – переключатель формы).

Сохранение каждой новой формы выполняется прежней командой *Ctrl+Shift+S*, либо командой *File/Save All*, или пиктограммой  (с именем *Save All*), расположенной на *Панели Инструментов*.

После исчезновения главной формы-заставки появится немодальная форма с текстом и двумя кнопками. Нажатие кнопки *Выход* обрабатывает функция *VuxodClick*, в тело которой впишите код, предназначенный для закрытия главной формы *Glav_Form* (и всего приложения):

```
Glav_Form->Close();
```


Щелчок по кнопке *Настройка* обрабатывает функция *NastrojkaClick*, её назначение состоит в показе модальной формы *Modal_Form*, что осуществляется функцией *ShowModal()*:


```
Modal_Form->ShowModal();
```

В нашем приложении *обе* вспомогательные формы порождаются при запуске программы, поэтому они *всегда* занимают оперативную память для своего размещения. Однако могут быть ситуации, когда при работе приложения отдельные его формы не всегда используются, поэтому их разумнее (например, для экономии оперативной памяти) порождать только когда в их существовании возникла необходимость. В таких случаях впереди предшествующего кода следует вписать операцию порождения формы (в нашем примере *Modal_Form*):

```
Application->CreateForm(__classid(TModal_Form), &Modal_Form);
```

Задачу порождения модальной формы выполняет метод *CreateForm()* неоднократно применяемого ранее указателя *Application* на класс *TApplication*. В функции *CreateForm()* используется ссылка на имя модальной формы *Modal_Form* и оператор *_classid*, в котором аргументом является имя типа этой же формы.

На модальной форме, помимо упомянутых ранее кнопок, расположите не-визуальные объекты *ColorDialog1* и *FontDialog1*, их заготовки находятся на вкладке *Dialogs* (диалоги), пиктограммы выглядят так:  (*ColorDialog*),

 (*FontDialog*). Назначение объектов такого типа состоит в том, чтобы вывести стандартные в *Windows* окна для выбора и установки параметров соответственно цвета и текста (имени, начертания, размера и цвета текста).

Щелчок по кнопке с именем *Cvet* обрабатывает функция *CvetClick*. В её тело впишем вначале такой код:

```
NeModal_Form->Label1->Font->Color= clRed;
```

Эта команда осуществит установку красного (*clRed*) цвета шрифта для текста, выведенного в объекте *Label1* немодальной формы *NeModal_Form*. Теперь предоставим пользователю приложения возможность выбора *произвольного* цвета с использованием объекта *ColorDialog1*. С этой целью указанную выше строку замените таким кодом:

```
if (ColorDialog1->Execute())  
    NeModal_Form->Label1->Font->Color= ColorDialog1->Color;
```

После установки выбранного цвета (нажатия кнопки *OK* в окне выбора цве-

тов) метод *Execute()* объекта *ColorDialog1* запишет в свойство *Color* для шрифта текста объекта *Label1* формы *NeModal_Form* то значение цвета, которое было выбрано в окне выбора *ColorDialog1*.

Нажатие кнопки с именем *Shrift* обрабатывает функция *ShriftClick*, в теле которой с использованием метода *Execute()* производится выбор параметров шрифта:

```
if (FontDialog1->Execute())
    NeModal_Form->Label1->Font->Assign(FontDialog1->Font);
```

Функция *Assign* является методом формы. Аргумент этой функции получает конкретное значение после нажатия кнопки *OK* на интерфейсе окна выбора шрифта в результате действия метода *Execute()*. Как отмечалось выше, среди параметров текста присутствует и его цвет, хотя возможности выбора цвета менее богаты, чем в случае применения *специального* окна выбора цвета, используемого в *CvetClick*.

Упомянутые и все другие диалоги являются невизуальными объектами, поэтому их местоположение на форме *не имеет значения* для вида интерфейса. При обращении к таким объектам вызываются стандартные диалоги, вид которых зависит от *Windows* и её настройки. При русифицированной версии *Windows* все надписи выводятся на русском языке, а при англоязычной – на английском.

Вызов любого диалога осуществляется методом *Execute()*. Если пользователь в диалоговом окне произвёл какой-то выбор, то эта функция возвращает *true*. При этом *все* выбранные параметры запоминаются в свойствах объекта-диалога и их можно прочитать и использовать для дальнейшей работы программы. Если же пользователь приложения нажал в диалоге кнопку *Отмена* или клавишу *Esc*, то функция *Execute()* возвращает *false*.

Для того чтобы главная форма *Glav_Form* могла породить немодальную форму *NeModal_Form* в её файл реализации следует включить заголовочный файл указанной немодальной формы (его имя *Ne_Mod_Form*) с применением директивы препроцессорной обработки *include*:

```
#include "Ne_Mod_Form.h"
```

Конечно, эту строку несложно записать (набрать) вручную. Однако *настоятельно* рекомендуем (во избежание ошибок) эту директиву включать при помо-

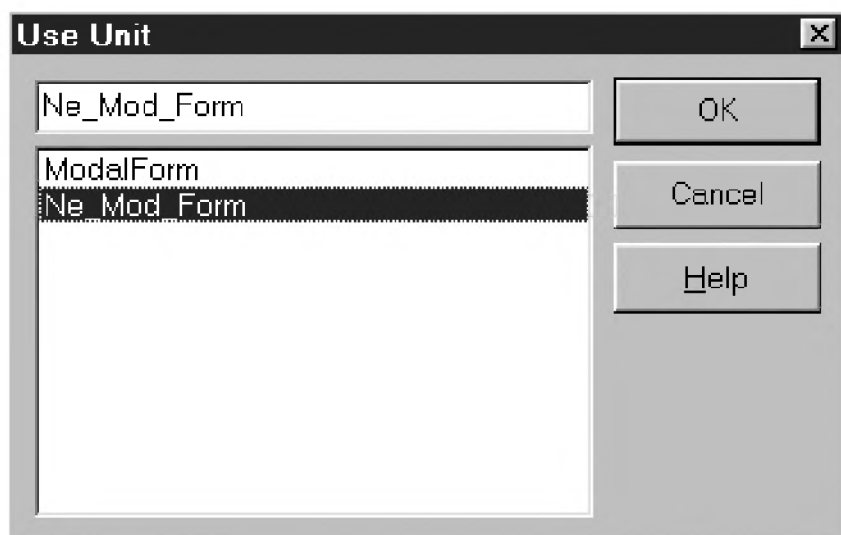


Рис. 11.9. Включение заголовочного файла *Ne_Mod_Form*.

щи команды *File/Include Unit Hdr..* (или *Alt+F11*). После ввода этой команды появится специальное окно (см. рис. 11.9), в котором можно выбрать нужный заголовочный файл и нажать клавишу *OK*.

В файле реализации немодальной формы (с именем *Ne_Mod_Form.cpp*) должны быть доступны главная *Glav_Form* (её файл реализации *Tri_Vida_Form.cpp*) и модальная *Modal_Form* (с файлом реализации *ModalForm.cpp*) формы, поэтому в начале файла *Ne_Mod_Form.cpp* включим заголовочные файлы упомянутых форм:

```
#include "ModalForm.h"//Заголовочный файл формы Modal_Form
#include "Tri_Vida_Form.h"//Заголовочный файл формы Glav_Form
```

Модальная же форма в нашем приложении должна «видеть» только немодальную форму *NeModal_Form* (и её объект *Label1*), поэтому в её файле реализации *ModalForm.cpp* запишем такую директиву компилятору:

```
#include "Ne_Mod_Form.h"//Заголовочный файл для NeModal_Form
```

Ниже приведены файлы реализации главной и подчинённых форм (кроме строк, которые генерирует *Builder* автоматически).

//Файл реализации главной формы

```
...
#include "Ne_Mod_Form.h"
void __fastcall TGlav_Form::FormCreate(TObject *Sender)
{
    Label1->Caption= "ТЕКСТ ПРИВЕТСТВИЯ";
    Timer1->Interval= 5000;//5 с
}//TGlav_Form::FormCreate

void __fastcall TGlav_Form::Timer1Timer(TObject *Sender)
{
    Glav_Form->Visible= false;// Скрываем Glav_Form
    NeModal_Form->Show();//Показываем NeModal_Form
    Timer1->Enabled= false;// Отключение Timer1
}//TGlav_Form::Timer1Timer
```

//Файл реализации немодальной формы

```
...
#include "ModalForm.h"
#include "Tri_Vida_Form.h"
void __fastcall TNeModal_Form::NastrojkaClick(TObject *Sender)
{
    //Немодальная форма будет создана и появится после нажатия
    //кнопки Настройка
    Application->CreateForm(__classid(TModal_Form), &Modal_Form);
    Modal_Form->ShowModal();
}//TNeModal_Form::NastrojkaClick

void __fastcall TNeModal_Form::VuxodClick(TObject *Sender)
{
    Glav_Form->Close();
}//TNeModal_Form::VuxodClick
```

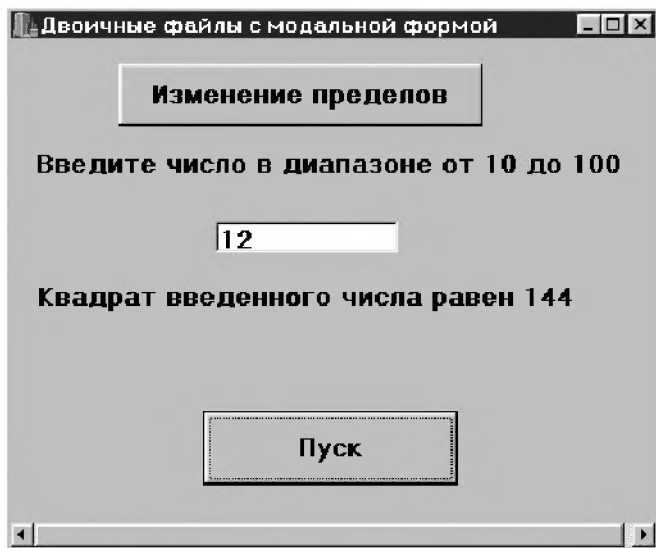


Рис. 11.10. Приложение *Двоичные файлы с модальной формой*

```
//Файл реализации модальной формы
...
#include "Ne_Mod_Form.h"
void __fastcall TModal_Form::ShriftClick(TObject *Sender)
{
    if (FontDialog1->Execute())
        NeModal_Form->Label1->Font->Assign(FontDialog1->Font);
} //TModal_Form::ShriftClick

void __fastcall TModal_Form::CvetClick(TObject *Sender)
{
    if (ColorDialog1->Execute())
        NeModal_Form->Label1->Font->Color= ColorDialog1->Color;
} //TModal_Form::CvetClick
```

Модернизация приложения Квадрат числа

Для закрепления правил работы с дополнительными формами модернизируем приложение *Квадрат числа*, которое разработано на девятом уроке (см. п. 9.4.4 и раздел 9.5). После переработки это приложение называется *Двоичные файлы с модальной формой*. Если ранее пределы возводимого в квадрат числа вводились с главной формы при помощи полей ввода, меню и кнопок, то теперь для этой цели используем модальную форму. Интерфейс приложения и его модальной формы показан на рис 11.10.

Имя основной формы – *Dvoich_f*, её файл реализации – *Dvoichnie_f.cpp*. Имя модальной формы – *Vvod_N_Min_N_Max*, файл реализации – *Modaln_Forma*. Изучите код приложения и его комментарии.

```
//Файл реализации главной формы
...
#include <math.h>
#include "Modaln_Forma.h" //Вписать или ввести при помощи
// команды File/Include Unit Hdr... или Alt+F11
AnsiString Rez_f;
const int n= 1000; //Максимальное число элементов массива
```

```
int N_Min= 10, //Минимальное вводимое число
    N_Max= 100; //Максимальное вводимое число
float fMas[n]; //Глобальный массив
AnsiString Fajl_Kvadratov = "Fajl_Kvadratov";
int iN; //Для запоминания введенного числа
float fR; //Для хранения квадрата введенного числа

void Fajl_Kvadr() //Создаёт двоичный файл, содержащий
                //квадраты чисел от N_Min до N_Max
{
    for (int ij = 0; ij< N_Max-N_Min; ++ij)
        fMas[ij]= pow(N_Min + ij, 2);
    int iFajl1;
    iFajl1= FileCreate(Fajl_Kvadratov);
    if (iFajl1== -1)
    {
        ShowMessage("Файл не удалось создать");
        Abort();
    } //if
    FileWrite(iFajl1, &fMas, sizeof(fMas));
    FileClose(iFajl1);
} //Fajl_Kvadr

void Slyga() //Считывает задаваемое число в переменную iN и
//записывает квадрат его значения из файла в переменную fR
{
    if ((iN< N_Min) || (iN> N_Max))
    {
        ShowMessage("Вы ввели число вне запрашиваемого диапазона");
        Abort();
    } //if
    int iFajl1;
    iFajl1= FileOpen(Fajl_Kvadratov, fmOpenRead);
    FileSeek(iFajl1, (int)sizeof(float)*(iN-N_Min), 0);
    FileRead(iFajl1, &fR, sizeof(fR));
    FileClose(iFajl1);
} // Slyga

void __fastcall TDvoich_f::PyskClick(TObject *Sender)
{
    //Запоминаем введенное число
    iN= StrToInt(Vvod_Chisla->Text);
    Fajl_Kvadr();
    Slyga();
    Rezyltat->Caption= "Квадрат введенного числа равен " +
        FloatToStrF(fR, ffFixed, 10, 0);
} // PyskClick

void __fastcall TDvoich_f::Izmenenie_PredelovClick(TObject *Sender)
{
    //Модальная форма создаётся и появляется после нажатия
    //кнопки Изменение пределов с именем Izmenenie_PredelovClick
    Application->CreateForm(__classid(TVvod_N_Min_N_Max), &Vvod_N_Min_N_Max);
    // Vvod_N_Min_N_Max – это имя модальной формы
} // Izmenenie_PredelovClick

void __fastcall TDvoich_f::FormCreate(TObject *Sender)
```

```

{
    Ykazanie->Caption= "Введите число в диапазоне от " +
        IntToStr(N_Min) + " до " + IntToStr(N_Max);
} // FormCreate

//Файл реализации модальной формы
...
extern int N_Min; //Минимальное вводимое число
extern int N_Max; //Максимальное вводимое число
//Служебное слово extern указывает на то, что переменные
//введены в другом модуле (файле), теперь они являются общими
//глобальными переменными обоих файлов

void __fastcall TVvod_N_Min_N_Max::ProdolgitjClick(TObject *Sender)
{
    N_Min= StrToInt(Vvod_N_Min->Text);
    N_Max= StrToInt(Vvod_N_Max->Text);
    Dvoich_f->Ykazanie->Caption= "Введите число в диапазоне от "
        +IntToStr(N_Min) + " до " + IntToStr(N_Max);

    Vvod_N_Min_N_Max->Close();
} // ProdolgitjClick

```

В отличие от главной, модальная форма в этом приложении порождается не автоматически при запуске программы, а только когда в ней возникнет необходимость: пользователь нажмёт кнопку *Изменение пределов*. Как отмечалось выше, этот способ создания вспомогательных форм можно рекомендовать для рачительного использования оперативной памяти. Задачу порождения модальной формы выполняет функция главной формы с именем *Izmenenie_PredelovClick*. С этой целью в ней применяется метод *CreateForm()* указателя *Application* на класс *TApplication*. В функции *CreateForm()* используется ссылка на имя модальной формы *Vvod_N_Min_N_Max* и оператор *_classid*, в котором аргументом является имя типа этой же формы:

```
Application->CreateForm(__classid(TVvod_N_Min_N_Max), &Vvod_N_Min_N_Max);
```

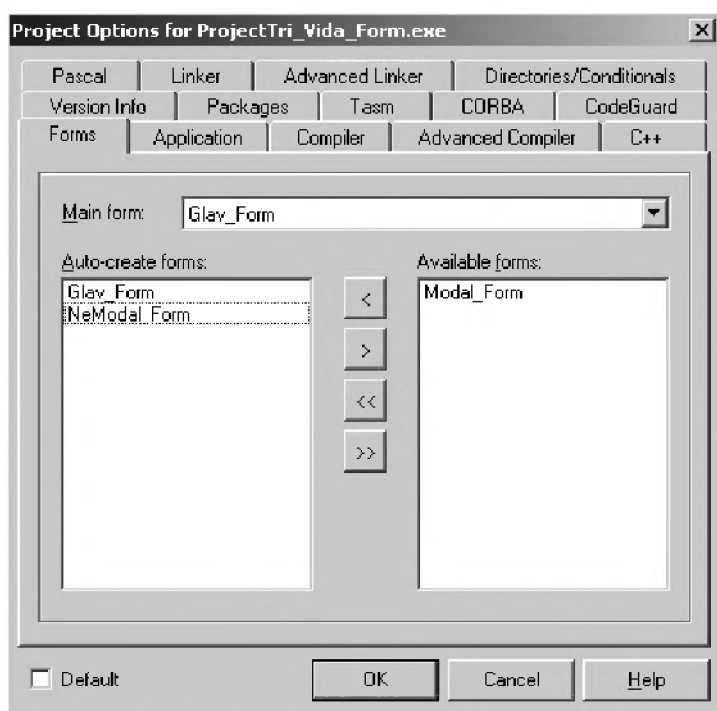


Рис. 11.11. Изменение свойств форм

Методы *Show()* и *ShowModal()* можно применять *только* к невидимой в данный момент форме. Поэтому в общем случае перед использованием упомянутых методов следует проверить свойство *Visible* формы. При выполнении методов *Show()* и *ShowModal()* возникает событие *onShow* формы. Оно возникает *до* момента видимости формы, поэтому обработку этого события можно использовать для настройки объектов открываемой формы. Если событие *onCreate* наступает для каждой формы *только один раз*, в момент её создания, то событие *onShow* происходит *всякий раз*, когда форма *делается видимой*.

Назначение статуса главной формы

Свойства форм, установленные по умолчанию в ходе их порождения, можно изменить при помощи страницы *Forms* опций проекта, представленной на рис.11.11.

Это окно выводится командой *Project/Options/Forms* (или *Ctrl+Shift+F11)/Forms*). В поле ввода с именем *Main form* можно назначить статус главной *любой* форме, имеющейся в проекте. Все формы из окна *Auto-create forms* (формы, создаваемые автоматически в момент запуска приложения) можно перемещать в окно *Available forms* (формы, доступные в приложении) и обратно при помощи соответствующих кнопок, расположенных между этими окнами. Формы со статусом *Available forms* порождаются лишь тогда, когда в их существовании возникает необходимость. При этом используется метод *CreateForm()* указателя на класс *TApplication*, применение которого иллюстрировалось в приложениях *Модальные и немодальные формы* и *Двоичные файлы с модальной формой*.

Вопросы для самоконтроля

- ☐ На сколько байт уменьшится оперативная память при объявлении трёх ссылок на конкретный объект? Как изменится эта величина, если ссылки объявить на объекты разных типов?
- ☐ Какое свойство ссылок используется в программировании наиболее часто?
- ☐ Сколько ссылок можно объявить на конкретный объект?
- ☐ Что произойдёт, если ссылке присвоить численное значение того же типа, что и тип ссылки?
- ☐ Какое предупреждение выведет *Builder*, когда тип ссылки не соответствует типу инициализирующей переменной либо объекта? Что оно означает?
- ☐ Почему ссылка не может иметь тип *void*?
- ☐ Можно ли при определении ссылки на константу в качестве инициализатора использовать константное выражение, тип которого не тождественен типу ссылки? Что при этом будет записано в ссылку на константу, если тип инициализирующего выражения является вложенным в тип ссылки?
- ☐ Что записывается в адресную часть указателя на ссылку?
- ☐ Зачем прибегают к использованию объектов типа *TGroupBox*, если на ин-

терфейсе располагается несколько наборов (групп) объектов типа *TRadioButton*?

- ☐ С какой целью применяются объекты типа *TCheckBox*?
- ☐ Укажите назначение флагов модальности в диалоговых окнах типа *TMessageBox*.
- ☐ Перечислите основные отличительные возможности объекта типа *TGauge* относительно объекта типа *TProgressBar*.
- ☐ Назовите метод, при помощи которого открываются объекты типов *TColorDialog*, *TFontDialog*, *TOpenDialog*, *TSaveDialog* и другие диалоги.
- ☐ Каким свойством задаётся интервал времени в объекте типа *TTimer*, после которого наступает событие *onTimer*? Зачем оно используется?
- ☐ В чём состоит основное различие между двумя существующими способами порождения вспомогательных форм?
- ☐ При каких условиях главную форму можно сделать невидимой?

11.7. Задачи для программирования

Задача 11.1. Разработайте программу, в которой используется функция, возвращающая ссылку на массив, все элементы которого получены преобразованием исходного массива, являющегося формальным параметром этой функции. Пусть преобразование исходного массива заключается в добавлении ко всем его элементам заданной константы.

Задача 11.2. Разработайте приложение, которое состоит из двух форм (с именами *Form1* и *Form2*). Первая форма является главной, играет роль логотипа или приветствия, появляется первой и находится на экране всего 5 секунд. На второй форме располагаются объекты *OpenDialog*, *SaveDialog* (вкладка *Dialogs*), *RichEdit*, *MainMenu*, *Button*. По команде *MainMenu Открыть* с объекта *OpenDialog* реализуйте возможность загрузки текстового файла в *RichEdit1*. После каких-либо изменений текста в *RichEdit1*, осуществите его сохранение (в формате *RTF*) в прежнем файле командой *Сохранить* и командой *Сохранить Как* – в файле с иным именем.

11.8. Варианты решений задач

Решение задачи 11.1

//Файл реализации задачи

...

const int n= 4;

const float fCislo= 0.7;

//Пользовательский тип на массив типа float из n элементов

typedef float float_Mas[n];

//Объявляем два массива типа float_Mas

float_Mas fMas={8.0, 6.0, 2004.0, 54.0},

fNov_Mas; //Этот массив используется для хранения

```

        //преобразованных элементов массива fMas
//Объявляем тип "ссылка на массив типа float"
typedef float_Mas &Ssul_Mas_float;

//Функция, возвращающая ссылку на преобразованный массив
Ssul_Mas_float Fyn_Ssul_Mas_float(float_Mas fMas, int iDlina)
{
    for (int ij= 0; ij< iDlina; ++ij)
        fNov_Mas[ij]= fMas[ij]+ fCislo;
    return &fNov_Mas;
} //Fyn_Ssul_Mas_float

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    float fX;
    String sY;
    for (int ii= 0; ii< n; ++ii)
    {
        fX= Fyn_Ssul_Mas_float(fMas, n)[ii];
        sY= FloatToStrF(fX, ffFixed, 10, 2);
        ShowMessage(sY);
    } //for_ii
} //Button1Click

```

Решение задачи 11.2

```

//Файл реализации главной формы

//Подключаем заголовочный файл для второй формы (Form2)
#include "Otkr_Soxran_Soxr_Kak.h"
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Label1->Height= 46; //Размер шрифта приветствия
    Label1->Caption= "Это текст приветствия";
    //Это можно сделать и в Инспекторе Объектов
    Timer1->Interval= 5000; // 5 секунд
} //FormCreate

void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    Form1->Visible= false; //Скрываем Form1
    Form2->Show(); //Показываем Form2
    Timer1->Enabled= false; //Отключаем таймер
} //Timer1Timer

//Файл реализации подчинённой формы
...
//Подключаем заголовочный файл для первой формы (Form1)
#include "Privetstvie.h"
void __fastcall TForm2::Button1Click(TObject *Sender)
{
    Form1->Close();
} // Button1Click
String Imja_Fajla= ""; //Глобальная переменная модуля

//Функция для обработки пункта меню Открыть
void __fastcall TForm2::N1Click(TObject *Sender)
{

```

```
if (OpenDialog1->Execute())
{
    // FileName – свойство OpenDialog1, предназначенное
    // для хранения имени файла
    Imja_Fajla= OpenDialog1->FileName;
    RichEdit1->Lines->LoadFromFile(Imja_Fajla);
} // if
} // N1Click

//Функция для обработки пункта меню Сохранить
void __fastcall TForm2::N2Click(TObject *Sender)
{
    //Если не вводилось новое имя файла
    if (Imja_Fajla != "")
        RichEdit1->Lines->SaveToFile(Imja_Fajla);
    else // Если вводилось новое имя файла
        if (SaveDialog1->Execute())
        {
            Imja_Fajla= SaveDialog1->FileName;
            RichEdit1->Lines->SaveToFile(Imja_Fajla);
        } // if
} // N2Click

//Функция для обработки пункта меню Сохранить как
void __fastcall TForm2::N3Click(TObject *Sender)
{
    SaveDialog1->FileName= Imja_Fajla;
    if (SaveDialog1->Execute())
    {
        Imja_Fajla= SaveDialog1->FileName;
        RichEdit1->Lines->SaveToFile(Imja_Fajla);
    } // if
} // N3Click
```

Урок 12. Символы и строки

12.1. Символьные переменные

Все символы, которые можно вывести на экран компьютера при помощи клавиатуры, сведены в специальную таблицу из 256 ячеек. В результате каждый символ приобрел свой номер или код, а сама таблица получила название: *таблица кодов символов*. В первую половину этой таблицы, в ячейки с кодами 0 – 127, входят символы из стандартного американского кода для обмена информацией – *ASCII* (*American Standard Code for Information Interchange*). В этой части таблицы, кроме арабских цифр, знаков пунктуации, строчных и прописных английских букв, находятся также служебные коды (0 – 31). Вторая часть таблицы кодов символов (ячейки с кодами 128 – 255) предназначена для кодировки *национальных* алфавитов, символов псевдографики и некоторых других знаков.

Объявляются символьные переменные при помощи служебного слова *char*, а их инициализация осуществляется с использованием одинарных кавычек:

```
char Ch= 'Ю';  
ShowMessage (Ch) ; //Ю
```

На экран выводится прописная буква Ю.

Вместо зарезервированного слова *char*, разрешается использовать идентификаторы-псевдонимы *Char* и *AnsiChar*. Приставка *Ansi* (здесь и ниже) происходит от национального института стандартов США (*American National Standards Institute*). Далее в пособии всегда используется *только* ключевое слово *char*. В памяти компьютера переменные типа *char* занимают ячейку объёмом в *один* байт. Однако в *C++* имеется тип данных *wchar_t*, предназначенный для хранения символов в *двухбайтовых* ячейках памяти. В *C++Builder* помимо этого служебного слова применяется его синоним *WideChar*.

Однако вернёмся к типу *char*. Для того чтобы узнать *код* любого символа (или номер символа (литеры)) в таблице кодов символов (не прибегая к справочнику) достаточно применить *явное* преобразование типа с использованием стандартного типа *unsigned char*:

```
char Ch= 'Ю';  
int Nomer= (unsigned char) Ch;  
ShowMessage (Nomer) ; //222
```

На экране появится число 222 – кодовый номер прописной буквы Ю.

Диапазон значений переменных типа *char*, выступающих в качестве чисел (см. ниже), простирается от –128 до 126, поэтому для *нумерации* ячеек таблицы символов он не очень подходит. Вместе с тем переменные типа *unsigned char* имеют диапазон, который *полностью* совпадает с диапазоном кодов символов: от 0 до

255. Именно по этой причине для преобразования символа в число необходимо использовать *только* тип *unsigned char*. При этом следует *особо* подчеркнуть, что числовой тип данных *unsigned char*, *не способен отображать* сами символы, с его использованием можно получить *только* код литеры. Поэтому применять тип *unsigned char* *вместо* типа *char* никак нельзя.

Далее приведём код, сообщающий о том, что строчная буква *ю* имеет номер 254:

```
char Ch= 'ю';
int Nomer= (unsigned char) Ch;
ShowMessage(Nomer); //254
```

Этим иллюстрируется правило, согласно которому *все строчные* буквы (как кириллицы, так и латиницы) имеют *большой* номер в таблице кодов, *чем прописные*.

Данные типа *char* – это одновременно и символы и их кодовые номера, поэтому букву *Я*, следующую за буквой *Ю* (в алфавите и кодовой таблице), можно вывести и при помощи такого фрагмента программы:

```
char Ch= 'Ю';
Ch= (unsigned char) (Ch + 1); //или Ch= (char) (Ch + 1);
ShowMessage(Ch); //Я
```

Если с символами обращаться как с числами, применять к ним арифметические операции, то *Builder* также воспринимает их как *числа*. В нашем примере к *номеру* символа *Ю* добавлена единица, а затем *новый номер* при помощи явного преобразования типа превращён в символ *Я*. В этом примере преобразование типа осуществимо и при помощи типа *char*.

Если служебные коды (0 – 31) использовать в качестве кодов *символов*, то выводится символ «|»:

```
char Ch= (unsigned char) (25);
ShowMessage(Ch); //|
```

Тип переменных *char* относится к *порядковому* типу. К этому типу относятся также *все* целочисленные типы, перечисления (см. тринадцатый урок) и некоторые другие типы данных. Данные *порядкового* типа имеют *конечное* число возможных значений, эти значения определенным образом упорядочены (поэтому тип называется *порядковым*) и каждое из значений имеет свой *порядковый* номер. Поэтому переменные типа *char* могут использоваться в цикле *for*.

```
String Stroka= "";
for (char Ch= 'A', Ch_Z= 'Z'; Ch<= Ch_Z; Ch++)
    if (Ch%4== 0)
        Stroka= Stroka + Ch + " " +
            IntToStr((unsigned char)Ch) + "\t" + "\n";
    else
        Stroka= Stroka + Ch + " " + IntToStr((unsigned char)Ch) + "\t";
RichEdit1->Text= Stroka;
```

Для реализации приведенного фрагмента программы (его можно поместить в функцию *PyskClick*, конструктор или функцию *FormCreate*), на форме следует расположить объект *RichEdit1*. Указанный фрагмент кода распечатывает в четыре колонки (в поле вывода окна *RichEdit1*) значения кодов *всех* латинских прописных букв. В этом фрагменте используется арифметическая операция *%* – ос-

таток от целочисленного деления, для перехода на новую строку применяется односимвольная константа «\n», а для вставки горизонтальной табуляции – односимвольная константа «\t». Эти и другие односимвольные константы называются *Esc-последовательностями*, все их значения сведены в таблице.

\n	перевод на новую строку
\t	ввод горизонтальной табуляции
\\	ввод обратного слеша
\'	ввод одинарной кавычки
\"	ввод двойной кавычки
\?	ввод вопросительного знака

12.1.1. Перевод символов DOS в символы Windows

Символы кириллицы, порождённые в редакторах *DOS*, например в *FAR*, называются символами *OEM*: *om original equipment manufacturer* – *изготовитель комплектного оборудования*. В текстовых редакторах *Windows* такие символы *искажены* (вместо требуемого символа кириллицы выводится иной символ). Для устранения искажений применяют две функции из библиотеки *Windows API*:

- ❑ *OemToChar(Isx_Txt, Rez_Txt)* предназначена для *перекодировки* полностью всего текста *Isx_Txt*;
- ❑ *OemToCharBuff(Isx_Txt, Rez_Txt, n)* – служит для преобразования только *первых n* символов из текста *Isx_Txt*.

Эти функции возвращают логический тип данных, совместимый с *bool*. Тип параметров *Isx_Txt* и *Rez_Txt* совместим с указателем на тип *char*. Первый параметр – это исходная строка, которую следует перекодировать, а второй – строка, предназначенная для занесения перекодированного текста. Параметр *n* в функции *OemToCharBuff* имеет целочисленный тип *DWORD* и определяет число символов, предназначенных для перекодировки (они отсчитываются от начала строки). Согласно описанию этой функции, *только* перекодированная часть исходного текста записывается во второй её текстовый параметр. У функции *OemToChar* имеется лишь два упомянутых выше текстовых параметра, изучим её на примере приложения: *Преобразовываем символы* (его интерфейс показан на рис. 12.1).

Текст файла *Isx_f.txt* создан в редакторе *FAR*, поэтому его фрагменты, написанные с использованием символов кириллицы, в окне объекта *TRichEdit1* (левое окно) искажены. После перекодировки функцией *OemToChar* этот же текст отображается правильно в окне объекта *TRichEdit2* (правое окно). Файл реализации этого приложения показан ниже.

```
TStringList *Spisok = new TStringList;
String Isx_f = "Isx_f.txt";//Исходный текст FAR
void Zagryzka (TStringList *Nabor_Strok, String Is_f)
{
    try
```

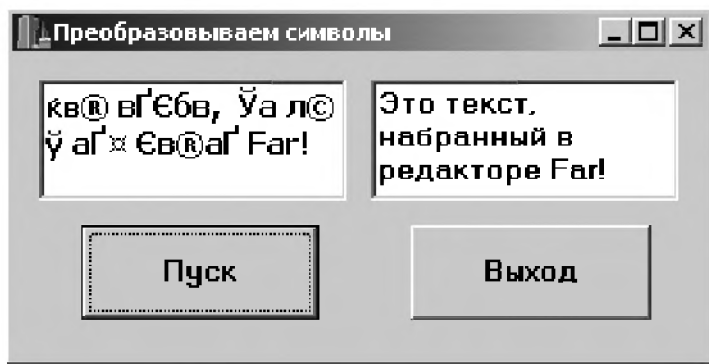


Рис. 12.1. Интерфейс приложения Преобразовываем символы

```

{
    Nabor_Strok->LoadFromFile(Is_f);
} //try
catch (...)
{
    ShowMessage("Файл \" " + Is_f + " \" не найден");
    Abort();
} //catch
} //Zagryzka

void __fastcall TDOSWindows::PyskClick(TObject *Sender)
{
    Zagryzka(Spisek, Isx_f);
    char *Stroka= new char; //Для временного хранения результата
    OemToChar(Spisek->Text.c_str(), Stroka);
    RichEdit1->Lines->LoadFromFile(Isx_f);
    RichEdit2->Text= Stroka;
    delete Stroka;
} //PyskClick

void __fastcall TDOSWindows::VuxodClick(TObject *Sender)
{
    delete Spisek;
    Close();
} //VuxodClick

```

Необходимо отметить, что в функции *PyskClick* строку

```
RichEdit2->Text= Stroka;
```

следовало бы заменить таким кодом

```
RichEdit2->Text= String(Stroka);
```

Однако и в прежней редакции приложение работает верно. Это объясняется тем обстоятельством, что свойство *Text* предназначено для хранения переменных типа *String*, а конструктор класса *String* (напоминаем, что в *Builder* тип *String* реализован как класс) позволяет преобразовывать к типу *String* переменные типа указатель на *char* (а также все числовые типы).

Первый параметр функции *OemToChar* преобразуется из переменной типа *String* в переменную типа указатель на *char* при помощи явного преобразования с использованием метода *c_str()*.

Для перекодировки только *части* текста применяется функция *OemToCharBuff*. Как отмечалось выше, последний её параметр указывает число преобразовываемых символов. Проиллюстрируем применение этой функции на прежнем приложении, в котором функцию *PyskClick* заменим такой:


```

void __fastcall TDOSWindowsN::PyskClick(TObject *Sender)
{
    DWORD N= 20; //Число преобразуемых символов
    Zagryzka(Spisok, Isx_f);
    char *Stroka= new char;
    OemToCharBuff(Spisok->Text.c_str(), Stroka, N);
    RichEdit1->Lines->LoadFromFile(Isx_f);
    //ќв® вҐЄбв, Ўа л© ў аҐ» Єв@aҐ Far!
    RichEdit2->Text= Stroka;
    //Это текст, набранныйќв® вҐЄбв, Ўа л© ў аҐ» Єв@aҐ Far!
    delete Stroka;
} //PyskClick

```

Функция *OemToCharBuff* должна возвращать (согласно своему описанию) *только заданное* число преобразованных символов. Однако такой режим работы осуществляется лишь, когда число преобразуемых символов не превышает $N=7$. При большем их числе часто, помимо перекодированных символов, выводится и оставшаяся (не преобразованная) часть текста. При этом в диапазоне значений $N=8-12$, выводится сообщение об ошибке. Если N превышает число символов исходного текста, то весь текст перекодируется верно. В чём здесь дело, мы не стали разбираться, поскольку задачи, решаемые упомянутой функцией, очень просто выполнить программно с привлечением функций по обработке строк (см. раздел 12.3).

12.1.2. Перевод символов Windows в символы DOS

Для преобразования символов *Windows* в символы *OEM* применяются функции из библиотеки *Windows API*: *CharToOem* и *CharToOemBuff*. Типы возвращаемых ими значений и типы их формальных параметров такие же, как и в функциях *OemToChar* и *OemToCharBuff*. Приложение, в котором применим функции *CharToOem* и *CharToOemBuff*, называется *Преобразование символов Windows в DOS*, его интерфейс показан на рис. 12.2. Имя левого окна вывода – *RichEdit1*, правого – *Memo1*.

Текст с кодировкой *ANSI* (*Windows*) набран в окне объекта *RichEdit1* (левое окно), это же сообщение после преобразования в символы *OEM* выводится в файл с именем *Rez_f.txt*. Если указанный файл открыть текстовым редактором *FAR*, то появится неискажённый исходный текст с кодировкой *OEM*. Этот результат работы программы достигается в два этапа. Вначале *весь* текст после его преобразования функцией *CharToOem* выводится в окно объекта *Memo1* (правое окно), как видно – он искажён, поскольку его кодировка отличается от кодировки, применяемой в *Windows*. Затем, используя метод *SaveToFile* этого объекта, происходит загрузка указанного текста в файл *Rez_f.txt*.

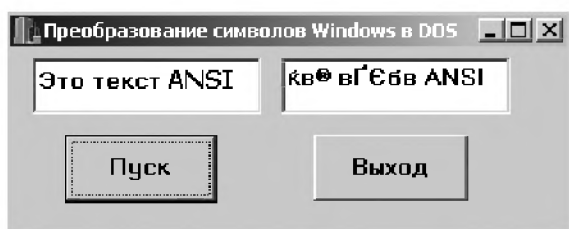


Рис. 12.2. Интерфейс приложения
Преобразование символов
Windows в DOS

Почему в настоящем приложении преобразованный текст выводится в объект типа *TMemo*, а не типа *TRichEdit* (как в прошлом приложении)? Дело в том, что объекты типа *TRichEdit* хранят информацию *только* в формате *RTF*, преобразуя *все* исходные форматы к указанному формату. Поэтому с использованием объекта упомянутого типа в файл *Rez_f.txt* загружается текст в формате *RTF*, который в редакторе *FAR* будет *искажённым*. Ниже приводится функция *PyskClick*, в которой выполняются описанные выше операции с использованием функции *CharToOem*.

```
void __fastcall TWinowsDOS::PyskClick(TObject *Sender)
{
    String Rez_f = "Rez_f.txt";
    char *Stroka= new char;
    CharToOem(RichEdit1->Text.c_str(), Stroka);
    Memo1->Text= Stroka;
    Memo1->Lines->SaveToFile(Rez_f);
    delete Stroka;
    Stroka= NULL;
} //PyskClick
```

Применение функции *CharToOemBuff* иллюстрируется новой функцией *PyskClick*, которой следует заменить в прежнем приложении одноимённую функцию. Функция *CharToOemBuff* выводит в файл *только заданное* число перекодированных символов. Если в окне *RichEdit1* набран текст *Преобразовываем символы*, то после нажатия кнопки *Пуск* в окне *Memo1* появится искажённый текст, а в файле *Rez_f.txt*, при его открытии редактором *FAR*, окажется выведенным *только* первые пять литер исходного текста: *Преоб*.

```
void __fastcall TWinowsDOS::PyskClick(TObject *Sender)
{
    DWORD N= 5; //Число преобразовываемых символов
    String Rez_f = "Rez_f.txt";
    char *Stroka= new char;
    //В RichEdit1 записан текст: Преобразовываем символы
    CharToOemBuff(RichEdit1->Text.c_str(), Stroka, N);
    Memo1->Text= Stroka; // Выводится искажённый текст
    Memo1->Lines->SaveToFile(Rez_f); //В файле записано: Преоб
    delete Stroka;
} //PyskClick
```

12.2. Массивы символов

12.2.1. Задание и инициализация

В *C++ Builder* разрешается использовать массив символов (элементами массива не могут быть *только* ссылки). Приведём пример объявления и инициализации переменных такого типа:

```
const int n= 6; //Число элементов массива
char Stroka[n]= "Привет";
ShowMessage(sizeof(Stroka)); //6
```

На экран будет выведено число 6 – число ячеек объявленного массива.

Присваивать массиву *char* заданную строку *целиком* можно *только* в ходе объявления массива, как показано в приведенном примере. *После* объявления массива *char* (с инициализацией или без инициализации какой-либо строковой константой) заполнять ячейки такого массива можно *только* посимвольно, например, с использованием одного из циклов:

```
for (int ij= 0; ij< n; ++ij)
    Stroka[ij]= 'Ф';
```

Число литер в присваиваемой константе (в нашем примере это слово *Привет*) должно быть *меньше* либо *равно* количеству ячеек задаваемого массива. Если же их окажется *больше* числа ячеек, то (как и в случае обычных массивов) *Builder* сообщит об ошибке: *E2225 Too many initializers* – слишком много инициализирующих значений.

Если количество символов инициализирующего выражения *меньше* числа запрашиваемых ячеек при объявлении массива, то *за последней* литерой окажутся неприсвоенные значения. Поэтому массив символов удобнее задавать и инициализировать так:

```
char Stroka[]= "Привет";
```

В этом случае *Builder* сам определит *фактическое* число элементов одномерного массива символов *Stroka*, произойдёт *автоматическое* задание длины массива по числу символов инициализирующей константы. Инициализирующее выражение в этом случае является *обязательным*.

Массив символов – это константный указатель на тип *char*. Поэтому строку текста можно задать также как указатель на тип *char*:

```
char *Yk_char= "всем!";
```

Следующие операторы последовательно выведут на экран слова *Привет* и *всем!*

```
ShowMessage(Stroka); //Привет
ShowMessage(Yk_char); //всем
```

Почему переменные типа *String* (аргумент функции *ShowMessage*, значения в свойствах *Text*, *Caption* и многих других) совместимы с переменными типа указатель на *char* и массив *char*, обсуждалось в предшествующем разделе.

Нумерация ячеек переменных типа указатель на *char* и массив *char* начинается с нуля, поэтому следующие операторы выведут на экран соответственно буквы «П» и «В».

```
ShowMessage(Stroka[0]); //П
ShowMessage(Yk_char[0]); //В;
```

Допустим следующий оператор присваивания:

```
Yk_char= Stroka;
```

Разрешены взаимные *поэлементные* присваивания для этих массивов, например:

```
Stroka[0]= Yk_char[1];
Yk_char[3]= Stroka[2];
```

После присваивания текстового значения переменным типа указатель на *char* или массив *char* (в случае автоматического определения длины массива) в ячейку, расположенную *после* последней литеры присвоенного текста, *записывается* нуль (символ нуля). Поэтому фактически переменные этих двух типов имеют на *одну* ячейку *больше*, чем число символов в присвоенном в тексте. В частности, в результате работы следующего фрагмента на экране покажется число 7:

```
char Stroka[] = "Привет";
ShowMessage(sizeof(Stroka)); //7
```

По этой же причине фрагмент кода, приведенный ниже, покажет на экране слово *нуль*.

```
char *Yk_char = "всем!";
if (Yk_char[5] == 0)
    ShowMessage("нуль");
//Не забывайте, что Yk_char[4] == '!'
```

Запись нуля (или нулевой *Esc*-последовательности «\0») в конце строки позволяет обозначить *границу* строковой переменной, что даёт возможность *программно*, перебирая символ за символом в строке, найти её конец.

12.2.2. Функции для обработки

Для обработки переменных упомянутых типов разработан целый ряд стандартных функций. Рассмотрим наиболее важные из них.

Функция *char* strcpy(char* Str_1, const char* Str_2)* обеспечивает копирование строки *Str_2* в строку *Str_1*, возвращает переменную *Str_1* (название происходит от слов *string* и *copy*). Вот пример её использования:

```
const int n = 16;
char *Yk_Ch_1, *Yk_Ch_2 = "Копируемый текст", Mas_Simv[n];
Yk_Ch_1 = strcpy(Mas_Simv, Yk_Ch_2);
ShowMessage(Yk_Ch_1);
ShowMessage(Mas_Simv);
```

На экране дважды появится сообщение: *Копируемый текст*. В этом и последующих примерах иллюстрируется *два* способа применения библиотечной функции. Это сообщение не изменится, если число *n* больше количества символов в сообщении. Первым фактическим параметром функции *strcpy* обязательно должен быть массив *char*, вторым фактическим параметром может служить указатель на *char* и массив *char*.

Функция *char* strcat(char* Str_1, const char* Str_2)* присоединяет строку *Str_2* в конец строки *Str_1*, возвращает переменную *Str_1*. Покажем её применение на таком примере:

```
char *Str_1 = "Начало", *Str_2 = " и конец";
St_1[20] = "Начало", St_2[10] = " и конец",
* Vozvr_Znach;
Vozvr_Znach = strcat(Str_1, Str_2);
ShowMessage(Str_1); ShowMessage(Vozvr_Znach); //Начало и конец
Vozvr_Znach = strcat(St_1, St_2);
ShowMessage(St_1); ShowMessage(Vozvr_Znach); //Начало и конец
```

На экране четыре раза появится сообщение: *Начало и конец*. В переменной *St_1*

умышленно зарезервировано *больше* число ячеек, чем требуется для размещения слова *Начало*. Это сделано для того, чтобы *суммарная* строка могла полностью разместиться в этой переменной. Однако в данном примере это число можно сделать равным даже шести – всё равно оба сообщения будут выведены на экран полностью. Но мы не рекомендуем злоупотреблять такими услугами компилятора. Как следует из примера, фактические параметры этой функции могут быть массивом *char* и указателями на *char*.

Можно делать *вложенные* вызовы этой функции. Вложенные вызовы позволяют соединять *несколько* строк. Предположим, требуется последовательно соединить строки *St_1*, *St_2*, *St_3*, а результат их слияния записать в четвертую переменную-строку *Rez*. Такая задача решается тремя вызовами функции *strcat*, вложенными друг в друга:

```
char St_1[] = "Один,", St_2[] = " два,", St_3[] = " три.",
Rez[20] = "", * Vozvr_Znach;
Vozvr_Znach = strcat(strcat(strcat(Rez, St_1), St_2), St_3);
ShowMessage(Rez); ShowMessage(Vozvr_Znach); //Один, два, три.
```

На экране дважды появится сообщение *Один, два, три*. Самый внутренний вызов функции *strcat* к пустой строке *Rez* присоединяет первую строку *St_1*. Второй, более внешний, вызов этой же функции к полученному результату добавляет строку *St_2*, после чего последний вызов функции *strcat* к полученной сумме строк присоединяет текст из переменной *St_3*. Если переменную *Rez* при её объявлении не инициализировать пустой строкой, то в ней могут находиться исходные, случайные значения (мусор), которые исказят результат сложения указанных выше трёх строк.

Гарантированно избавиться от исходного текста в переменной *Rez* можно также при помощи функции *strcpy*, которой следует заменить самый внутренний вызов функции *strcat*:

```
Vozvr_Znach = strcat(strcat(strcpy(Rez, St_1), St_2), St_3);
```

Для поиска *первого* вхождения строки *Str_2* в строку *Str_1* применяется функция *char* strstr(const char* Str_1, const char* Str_2)*. При успешном поиске эта функция возвращает указатель на первый символ искомого текста в строке поиска. Если же в строке поиска нет искомого текста, то функция *strstr* возвращает *NULL*.

Проиллюстрируем использование функции *strstr* в задаче поиска в тексте заданного слова и замене его другим словом. В приведенном ниже фрагменте используется функция *strlen(const char * Str)*. Она возвращает число литер своего аргумента *Str* без учёта нулевого конечного символа.

```
//Это исходное предложение
char Isx_Str[] = "Приветствую вас, товарищи!",
Poisk[] = "товарищи", Zamena[] = "господа", Rez[60] = "", *Yk;
Yk = strstr(Isx_Str, Poisk);
//Yk указывает на первый символ строки Poisk в исходной //строке Isx_Str
if (Yk) //Если Yk != 0
{ //В объект указателя Yk записывается нуль, этим
*Yk = 0; //выделяется часть строки до уничтожаемого текста,
//поскольку произвольная строка заканчивается нулём
```

```
//Сдвигаем Yk на длину заменяемого текста
Yk += strlen(Poisk);
//Теперь Yk указывает на первый символ после
//заменяемого слова
Yk= strcat(strcat(strcpy(Rez, Isx_Str), Zamena ), Yk);
ShowMessage(Yk);
ShowMessage(Rez);
} // if
else
ShowMessage("Текст не найден");
```

В начале этого фрагмента в переменную *Yk* заносится либо нуль (если искомого слова в предложении нет) либо адрес *первой* литеры найденного слова в исходном предложении. В первом случае появится сообщение: *Текст не найден*. При успешном поиске вместо первой литеры найденного слова записывается нуль. Это позволяет вызовом функции *strcpy(Rez, Isx_Str)* в переменную *Rez* скопировать только начальную часть строки (до найденного слова). Затем внутренний вызов функции *strcat* к этому фрагменту первоначального текста (*Приветствую вас,*) добавит текст из переменной *Zamena*. Поскольку *Yk* перемещается на литеру, расположенную *за* последним символом удаляемого слова (*Yk += strlen(Poisk)*), то внешний вызов функции *strcat* добавляет к скорректированной части текста часть исходного предложения, расположенную после заменяемого слова (!).

Функция *strcat* возвращает адрес первого символа своего первого аргумента *Rez*, поэтому этот адрес записан в переменную *Yk*, после чего выводится на экран её содержимое при помощи функции *ShowMessage: Приветствую вас, господа!* Этот же результат достигается, когда аргументом функции *ShowMessage* является переменная *Rez*.

Описанные выше функции не рекомендуем использовать в своей практической деятельности, поскольку в *C++Builder* введены значительно более удобные функции для решения тех же задач. Это касается и многих других функций *C++*, предназначенных для обработки переменных типа указатель на *char*. Однако функция *strtok* не имеет своего двойника в *C++Builder*. Вместе с тем она очень полезна, уже использовалась на девятом уроке, и будет применяться на этом и последующих уроках. Поэтому разъясним и проиллюстрируем её возможности более детально.

Функция *char* strtok(char* Str_1, const char* Str_2)* ищет в строке *Str_1* первое вхождение разделителей, указанных в строке *Str_2*, после чего в месте обнаруженного разделителя усекается содержимое переменной *Str_1*, затем возвращается усечённая часть исходной строки (до литеры отсечки). В качестве разделителей могут служить пробел, запятая, точка, другие знаки пунктуации и *любые* иные символы. При повторном вызове этой функции, если в качестве первого её параметра указать *NULL*, она обрабатывает *оставшуюся*, отсечённую, часть строки: усекает её по новому вхождению разделителя, приведенного в списке разделителей. Такие свойства функции *strtok* позволяют исходную строку разбить на части, находящиеся между разделителями. Проиллюстрируем эту воз-

возможность на простом примере.

```
char Isx_Tekst[70]= "Первое, второе, третье слово.",
    Razdeliteli[]= " ,.",
    *Yk;
Yk= strtok(Isx_Tekst, Razdeliteli); //Выделено первое слово
if (Yk) //Yk!= 0, в Yk записано первое слово
    ShowMessage(Yk);
while (Yk) //Пока Yk не станет указывать на нулевой символ,
    //записанный в конце строки
{
    Yk= strtok(NULL, Razdeliteli);
    if (Yk)
        ShowMessage(Yk);
} // while
```

Здесь в исходном тексте слова перечислены через запятую и пробел, а в конце предложения поставлена точка. Поэтому в список разделителей включены только пробел, запятая и точка. Первый вызов функции позволил с использованием оператора *if* и функции *ShowMessage* вывести первое слово исходного текста. Далее при помощи цикла производится дальнейшее последовательное усечение оставшейся части строки и вывод её слов. Обращаем внимание на то, что сами разделители (пробел, запятая, точка) не выводятся. На экране появится такая последовательность слов:

*Первое
второе
третье
слово*

В настоящем подразделе рассмотрены функции *strcpy*, *strcat*, *strstr*, *strlen* и *strtok*. Все эти библиотечные функции находятся в заголовочном файле *string.h*, который к любому приложению подключается по умолчанию.

12.3. Переменные типа *String*

Как отмечалось на предшествующих уроках, в *C++Builder* тип строк *String* (*AnsiString*) реализован как класс. Подробно о классах рассказывается на пятнадцатом уроке, поэтому сейчас ограничимся лишь краткой информацией о них. Класс – это совокупность данных и методов (функций) их обработки. При использовании указателей на классы доступ к данным и методам экземпляра класса осуществляется при помощи операции стрелка «->». Такая операция использовалась ранее для доступа к свойствам и методам визуальных объектов интерфейса приложений.

Доступ к методам класса *String* осуществляется непосредственно через объекты – экземпляры этого класса, а не через указатели на них. Поэтому для вызова метода класса *String* применяется операция точка «.». Вызов метода для указателя на тип (класс) *String* осуществляется при помощи операции стрелка. Например, вот так вызывается функция рассматриваемого класса, предназначенная для определения количества символов в строке:

```
String Stroka= "Привет", *Strok= &Stroka;
ShowMessage(Stroka.Length()); //Доступ через объект
ShowMessage(Strok->Length()); //Доступ через указатель
```

На экране дважды появится число 6. На самом деле в строках такого типа имеет ещё и конечный нулевой символ, однако метод *Length()* его не учитывает, точно так же как и функция *strlen*, предназначенная для обработки строк типа указатель на *char* (см. предшествующий раздел).

При объявлении переменные типа *String* инициализируются пустой строкой. Это лишь одно из *многих* других достоинств переменных этого типа по сравнению с массивом *char* и указателей на тип *char* (далее также *char**). Переменным типа *String* после их инициализации можно присваивать как строки *большой* (вплоть до $\sim 2^{31}$ символов), так и *меньшей* длины, чем длина первоначальной инициализируются строки. При этом в последнем случае никаких проблем с «мусором» (неприсвоенными значениями) не возникает, поскольку переменная этого типа *автоматически* уменьшается (или увеличивается) до *фактического* количества символов во вновь присвоенном тексте.

Вместе с тем имя экземпляра класса типа *String* – это не что иное, как указатель, поэтому следующий фрагмент кода выведет на экран число 4 – объём ячейки для хранения адреса объекта.

```
String Stroka= "Привет";
ShowMessage(sizeof(Stroka)); //4
```

Таким образом, минимальный объём памяти, выделяемый для хранения таких переменных равен четырём байтам (указатель на пустую строку). Максимальный же объём – огромен, он составляет около 2 Гбайт.

Обращение к отдельным литерам строки осуществляется также как и в массивах – через операцию квадратные скобки (это один из наиболее ясных способов):

```
ShowMessage(Stroka[1]); //П
```

При этом нумерация литер строки начинается с *единицы*, а не с нуля, как в массиве *char* и указателе на тип *char*. Поэтому в приведенном примере на экран выводится буква *П*. Обращаться к *нулевой* ячейке таких строк *запрещено*.

Изучаемый в настоящем разделе тип данных использовался на *всех* уроках, например, в свойствах *Text* и *Caption* различных визуальных объектов. Поэтому ниже лишь напомним функции для взаимного преобразования числовых и текстовых данных: *IntToStr*, *FloatToStr*, *FloatToStrF*, *StrToInt*, *StrToFloat*. Помимо упомянутых функций имеются ещё и функции, назначение которых во многом ясно из их названия: *ToDouble*, *ToInt*, *CurrToStr*, *CurrToStrF*, *ToIntDef*.

Здесь *Curr* происходит от *Currency* – это вещественный тип данных, предназначенный для наиболее точных, монетарных (банковских) вычислений. Переменные этого типа находятся в интервале $-922\,337\,203\,685\,477,580\,8$ – $922\,337\,203\,685\,477,580\,7$, для их вывода используется 19 – 20 разрядов, они занимают объём памяти в 8 байт.

Функция *CurrToStr* (*Chislo*) преобразует число *Chislo* типа *Currency* к текстовому типу и после математической точки оставляет до *четырёх* цифр.

Функция *CurrToStrF* (*Chislo*, *Format*, *Cifru*) соответствует функции *FloatToStrF* с 19 цифровыми разрядами. Параметр *Chislo* – это величина типа *Currency*, *Format*

Функции *CurrToStr* и *CurrToStrF* предназначены для обработки переменных типа *Currency*, однако они не принадлежат классу *String*, не являются его методами (аналогично *IntToStr*, *FloatToStr*, *FloatToStrF*, *StrToInt*, *StrToFloat*). Вместе с тем функции *ToInt* и *ToDouble* являются методами класса *String*, поэтому их вызов осуществляется так:

Существует ещё целый ряд *других* функций, предназначенных для взаимного преобразования числовых и текстовых переменных. В этих функциях используются различные типы форматирования текстового вида чисел. Однако не будем отвлекаться на их рассмотрение, займёмся изучением *более* важных функций класса *String*.

```
String Stroka= "Привет";  
char *Ch= Stroka.c_str();
```

```
String Str_1= "Раз, ", Str_2= "два, ",
      Str_3= "три, ", Str_4= "четыре, ",
      Str_5= "пять", Str_6= "вышел зайчик погулять.",
      Rez;
Rez= Str 1 + Str 2 + Str 3 + Str 4 + Str 5 + Str 6;
```

Оператор конкатенации имеет *более высокий* приоритет по отношению к операциям отношений: $<$, $<=$, $>$, $>=$, $!=$, $==$. Поэтому в следующем примере на экран

будет выведено сообщение: *Ку-ку*.

```
String Str_1= "AA", Str_2= "AA", Str_3= "AAAAAAA";
if (Str_1 + Str_2< Str_3)
    ShowMessage("Ку-ку");
```

12.3.1. Сравнение строк

Задайте четыре текстовые переменные:

```
String Str_1= "Яблоко",
        Str_2= "Яблоко",
        Str_3= "Яблоко ",
        Str_4= "яблоко";
```

Теперь проанализируйте результаты сравнения следующих строк:

```
Str_1== Str_2 //Все символы переменных совпадают
Str_3> Str_1 //Любой символ всегда больше пустого места
Str_1< Str_4 //т. к. (unsigned char)'Я'< (unsigned char)'я'
```

Такие заключения обусловлены следующими правилами, используемыми при сравнении строк:

- ☐ Выполнять сравнение строк посимвольно, начиная с первого символа в каждой строке.
- ☐ Если окажется, что сравниваемые строки имеют одинаковую длину и посимвольно эквивалентны, то тогда одна строка равна другой.
- ☐ Если в результате посимвольного сравнения код символа одной строки окажется больше (меньше) кода символа другой строки, то и строка, в которой он находится, считается большей (меньшей). Оставшиеся литеры строк и общая длина сравниваемых строк в этом случае на результат сравнения влияния не оказывают.

12.3.2. Стандартные функции для обработки строк

Познакомимся с рядом других наиболее важных функций, предназначенных для обработки переменных типа *String*. Часто возникает необходимость определить в исследуемой строке номер литеры, с которой начинается искомый фрагмент текста (например, заданное слово). В частности, это требуется для уничтожения заданного фрагмента или замены его другим текстом. При решении таких и аналогичных задач применяется функция *int Pos(const String& PodStroka)*, являющаяся методом класса *String*. Следующий фрагмент программы выведет на экран число 8 – позицию первой литеры слова *весна* в переменной *Stroka*.

```
String Stroka= "Пришла весна!",
        PodStroka= "весна";
int iPosic = Stroka.Pos(PodStroka); //Позиция буквы в
ShowMessage(iPosic); //8
```

Если в строке *Stroka* искомый текст *PodStroka* отсутствует, то метод *Pos* возвращает нулевое значение.

Для устранения из строки *Stroka*, начиная с заданной позиции *iPosic*, фраг-

```
String Stroka= "Пришла весна!",  
    PodStroka= "весна";  
int iPosic= Stroka.Pos(PodStroka),  
    iDlina = PodStroka.Length();//Длина удаляемого слова  
//Удаляем iDlina литер, начиная с позиции iPosic  
Stroka.Delete(iPosic, iDlina);  
ShowMessage(Stroka); //Пришла!
```

Теперь вставим в изменённый текст *Stroka* слово *зима* (*NovPodStroka*), начиная с позиции *iPosic*, с которой начиналось ранее слово *весна*. С этой целью используем ещё один метод класса *String*: *String& Insert(const String& NovPodStroka, int iPosic)*:

Методы *Insert* и *Delete* возвращают ссылку на модернизированную строку и производят изменение строки, для которой они вызываются. В приведенном коде корректно обработана ситуация, когда искомый текст *PodStroka* в строке *Stroka* не найден и метод *Pos* возвращает ноль.

Выше были рассмотрены очень важные функции, с использованием которых можно решить *большинство* задач по обработке текстовых переменных. Удобство работы с ними и ясность получаемого кода легко можно оценить, сопоставляя приведенные примеры с решениями аналогичных задач, в которых применяются функции обработки переменных типа *char** (см предшествующий раздел). Ниже познакомимся ещё с несколькими функциями, знание которых *существенно* облегчает решение задач, приводит к тому, что код приложения становится короче и яснее. Создатели *C++Builder* разработали целую серию новых удобных функций, удачно дублирующих уже существующие функции *C++*. Среди таких функций имеются и функции, которые обрабатывают переменные типа *char**. Все они находятся в библиотеке *SysUtils.hpp*, которая также как и заголовочный файл *string.h* к каждому приложению подключается *автоматически*.

Следует заметить, что все эти функции (для переменных *String* и *char**) можно разработать самостоятельно, создать *свои*, пользовательские, функции обработки строк. Однако ввиду существования богатых библиотек стандартных функций такая работа оправдана лишь в учебных целях. Таким образом, при решении задач рекомендуем не спешить с разработкой своих функций. Вначале попытайтесь найти подходящую *стандартную* функцию в библиотеке *SysUtils.hpp*. Только при неудачном поиске требуемой функции в этом файле стоит обращаться к библиотеке *string.h*. Если и в этой библиотеке подходящей функции нет, тогда уж разрабатывайте свою функцию.

Итак, знакомимся с другими полезными функциями (методами) класса *String*. Методы *String LowerCase()* и *String UpperCase()* *возвращают* строку, в которой все символы преобразованы соответственно к нижнему и верхнему регистрам. При этом их применение *не влияет* на исходную строку *Stroka*:

```
String Stroka= "ПрИвЕт!";
ShowMessage(Stroka.LowerCase()); //привет!
ShowMessage(Stroka.UpperCase()); //ПРИВЕТ!
ShowMessage(Stroka); // ПрИвЕт!
```

Для осуществления проверки строки *Stroka* на наличие в ней каких-либо символов применяется метод *IsEmpty()*, возвращающий *true*, если строка пуста:

```
String Stroka;
if (Stroka.IsEmpty())
    ShowMessage("Строка пуста!"); // Строка пуста!
```

В ходе решения задач (см., например, раздел 12.5), связанных с обработкой текстовых переменных, часто требуется, чтобы *каждая* строка многострочного текста начиналась и заканчивалась *одним* пробелом. Для достижения этой цели вначале устраняют все пробелы слева и справа каждой строки, а затем операцией конкатенации добавляют по одному пробелу в начало и конец строки. Проиллюстрируем методы, которые используются для этих целей.

```
String Stroka= "    П    ";
ShowMessage("!" + Stroka.TrimLeft() + "!"); //!П    !
ShowMessage("!" + Stroka.TrimRight() + "!"); //!    П!
ShowMessage("!" + Stroka.Trim() + "!"); //!П!
ShowMessage("!" + Stroka + "!"); //!    П    !
```

Как очевидно из названия методов и примеров их использования методы *String TrimLeft()*, *String TrimRight()*, *String Trim()* *возвращают* текст обработанной строки *Stroka*, в котором соответственно устранены все пробелы слева, справа и с обоих его концов (*trim* – подстригать, подрезать, обрезать кромки). При этом текст исходной строки *не изменяется* (см. последнюю строку примера).

Уменьшить длину строки *Stroka* до требуемого количества символов *in* возможно при помощи метода *void SetLength(int in)*. Длина исходной строки не уменьшается, если её длина *меньше in*.

```
String Stroka= "Длина строки",
    Dlina= "Длина";
int in= Dlina.Length(); //Длина новой строки
ShowMessage(Stroka.SetLength(in)); //Длина
ShowMessage(Stroka); //Длина
```

```
char Ch= 'Ф';//Символ, который требуется повторить
int in= 5;//Число повторений символа
String Stroka= String::StringOfChar(Ch, in);
ShowMessage(Stroka);//ФФФФФ
```

```
String Stroka= "Вырежем второе слово",
    Vurezka;
int iPosic= 9, iDlina= 6;
Vurezka= Stroka.SubString(iPosic, iDlina);
ShowMessage(Vurezka); //второе
ShowMessage(Stroka); //Вырежем второе слово
```

Примером такой *интеллектуальной* функции служит, в частности, функция *StringReplace*. Она способна заменить первое или *все* вхождения заданного фрагмента текста *Isxodn_Fragment* на новый фрагмент *Novuj_Fragment* в строке *Isxodn_Stroka*. При этом она может учитывать или не учитывать регистр литер в заменяемом фрагменте *Isxodn_Fragment*. Все указанные её параметры типа *String* являются константными, не изменяющимися при работе функции. Эта функция *возвращает* изменённое значение аргумента *Isxodn_Stroka*. Четвёртым её параметром является флаг-множество *TReplaceFlags()*, указывающий как производить замену. Если в этот флаг-множество при помощи операции «<<» поместить значение *rfReplaceAll*, то заменяться будет не только первое вхождение фрагмента, но и все другие его вхождения. Помещение во множество значения *rfIgnoreCase* позволяет игнорировать регистр литер искомого фрагмента *Isxodn_Fragment*. Указанные параметры можно помещать во множество по одному, либо оба в *любой* последовательности. Подробнее о множествах рассказывается на следующем уроке.

```
String Isxodn_Stroka, Izmenen_Stroka,
        Isxodn_Fragment, Novuj_Fragment;
//Содержание исходного текста
Isxodn_Stroka= "Здравствуйтe, ТовaРищи! Дорогие товарищи!";
```

```

Isxodn_Fragment= "товарищи";//Заменяемый фрагмент текста
Novuj_Fragment= "господа";//Текст замены
Izmenen_Stroka= StringReplace(Isxodn_Stroka,
                              Isxodn_Fragment, Novuj_Fragment,
                              TReplaceFlags()<<rfIgnoreCase<<rfReplaceAll);
ShowMessage(Izmenen_Stroka);
//Здравствуйтесь, господа! Дорогие господа!
ShowMessage(Isxodn_Stroka);
//Здравствуйтесь, ТоВаРиЩи! Дорогие товарищи!

```

Функция *IsDelimiter(const String Razdeliteli, const String Stroka, int iNom)* отвечает на вопрос о том, является ли символ с номером *iNom* в строке *Stroka* одним из символов, указанных в строке *Razdeliteli*. Применение этой функции позволяет заменять требуемые символы строки. Ниже в строке *Stroka* все символы *верхние одиночные кавычки* (этот символ указан в строке *Razdeliteli*) заменяются *верхними двойными кавычками*.

```

String Stroka= "Газета 'Известия'",
Razdeliteli= "'";
for (int ij= 1; ij<= Stroka.Length(); ij++)
    if(IsDelimiter(Razdeliteli, Stroka, ij))
        Stroka[ij]= '"';
ShowMessage(Stroka); // Газета "Известия"

```

В качестве разделителей могут использоваться *любые* символы, а не только знаки табуляции, пробелы, кавычки и знаки пунктуации. В следующем фрагменте все символы цифр в строке *Stroka*, кроме 0, будут уменьшены на 1.

```

String Stroka= "Проверка: 0123456789",
Razdeliteli="123456789";
for (int ij= 1; ij<= Stroka.Length(); ij++)
    if(IsDelimiter(Razdeliteli, Stroka, ij))
        Stroka[ij]-= 1;
ShowMessage(Stroka); // Проверка: 0012345678

```

Такое преобразование оказалось возможным вследствие того, что с типами *char* (*AnsiChar*) можно обращаться как с целыми числами и как с символами – все зависит от контекста (см. раздел 12.1). Поскольку над символами строки выполняются арифметические действия, поэтому они воспринимаются как числа (коды символов). Функция *ShowMessage* воспринимает символы строки *Stroka* вновь как символы, а не коды.

Функция *String WrapText(const String Isxodn_Stroka, const String Razbivka, const TSysCharSet &Simvolu_Razdeliteli, int in)* позволяет разбить исходный текст *Isxodn_Stroka* на последовательность строк (или на части, разделённые знаками табуляции). Разбиение текста *Isxodn_Stroka* происходит по разделителям, символы которых *заранее* помещаются во множество *Simvolu_Razdeliteli*. Разбивка исходного текста может осуществляться знаком табуляции «\t» (одним или более) или знаком «\n» перехода на новую строку (см. раздел 12.1), для чего указанные *Esc*-последовательности следует занести в переменную *Razbivka*. Параметр *in* задаёт *максимальное* количество символов в строках возвращаемого этой функцией текста (в примере – это переменная *Preobraz_Stroka*). Если в исходном тексте

встречаются слова, у которых больше, чем *in* символов, то эти слова не разделяются.

```
TSysCharSet Simvolu_Razdeliteli; // Объявление множества
//Помещение символов-разделителей во множество
Simvolu_Razdeliteli<< ' '<<'.'<<','<<','<<';
String Isxodn_Stroka, Preobraz_Stroka, Razbivka= "\n";
Isxodn_Stroka= "Первое, второе и третье слово. Слово_более_15_символов.";
Preobraz_Stroka= WrapText(Isxodn_Stroka, Razbivka, Simvolu_Razdeliteli,
15);
ShowMessage(Preobraz_Stroka);
```

Исходный текст разбивается на строки до 15 символов. В этом примере разбиение производится после пробелов и знаков пунктуации. Слова с длиной более 15 символов не разрываются. На экране появится текст:

*Первое, второе
и третье слово.
Слово_более_15_символов.*

Как видно из примера, символы-разделители *остаются* в преобразованном тексте.

12.3.3. Иллюстрация обработки строк

Приложение Удаление лишних пробелов

Проиллюстрируем применение отдельных стандартных функций для обработки строк в приложении *Удаление лишних пробелов*. Его интерфейс приведен на рис. 12.3. Объекты типа *TEdit* интерфейса, предназначенные для ввода имён файлов с исходным и результирующим текстами, имеют соответственно имена *Isx_Fajl* и *Rez_Fajl*.

В текстовом файле *Isx_f.txt* находится текст, состоящий из нескольких строк, слова впереди и позади себя имеют один и более пробелов. Программа преобразовывает исходный текст таким образом, чтобы в начале и конце *каждого* слова осталось только *по одному* пробелу. В файле *Rez_f.txt* после соответствующих комментариев помещаются начальный и преобразованный тексты. Другие пояснения к работе приложения приведены после его кода.

Код приложения Удаление лишних пробелов

```
...
TStringList *Spis_Isx= new TStringList,
               *Spis_Rez= new TStringList;
String Isx_f, Rez_f;
void Vvod_Spiska(int & iSt) //iSt - число строк текста
{
    try
    {
        Spis_Isx->LoadFromFile(Isx_f);
    } //try
    catch (...)
    {
```

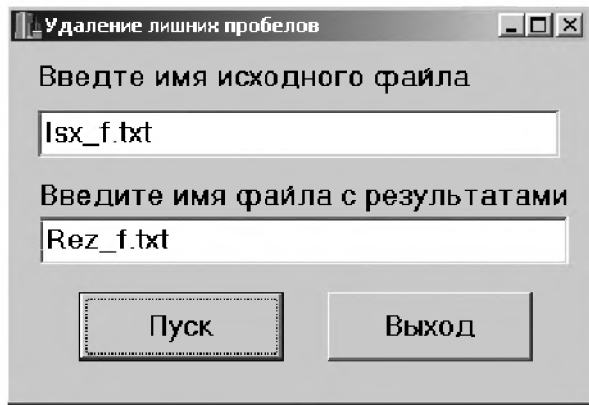


Рис. 12.4. Интерфейс приложения Удаление лишних пробелов

```

    ShowMessage("Файл \" " + Isx_f + " \" не найден");
    Abort();
} // catch
iSt = Spis_Isx->Count; // Число строк исходного текста
} // Vvod_Spiska

void Vuvod_Spiska(int iSt) // Создает копию текста
{
    for (int ij = 0; ij < iSt; ++ij)
        Spis_Rez->Add(Spis_Isx->Strings[ij]);
} // Vuvod_Spiska

void Ydal_Prob(int iSt) // Удаляет лишние пробелы
{
    for (int ij = 0; ij < iSt; ++ij)
    {
        Trim(Spis_Isx->Strings[ij]);
        Spis_Isx->Strings[ij] = " " + Spis_Isx->Strings[ij] + " ";
        int ii = Spis_Isx->Strings[ij].Pos(" ");
        while (ii > 0)
        {
            // Функция Delete возвращает указатель на
            // модернизированную строку
            // Уничтожили один пробел
            Spis_Isx->Strings[ij] =
                Spis_Isx->Strings[ij].Delete(ii, 1);
            // Положение следующих двух пробелов
            ii = Spis_Isx->Strings[ij].Pos(" ");
        } // while
    } // for_ij
} // Ydal_Prob

void __fastcall TYdal_Lish_Prob::PyskClick(TObject *Sender)
{
    int i_n; // Фактическое число строк исходного текста
    Isx_f = Isx_Fajl->Text; // Ввод имён файлов
    Rez_f = Rez_Fajl->Text;
    Vvod_Spiska(i_n);
    // Добавляем новую строку списка
    Spis_Rez->Add("Исходный текст с лишними пробелами");
    Vuvod_Spiska(i_n);
    Spis_Rez->Add(""); // Пропуск строки
    Spis_Rez->Add("Текст после обработки:");
    Ydal_Prob(i_n);
    Vuvod_Spiska(i_n);
}

```



```
//Создаём файл Rez_f и записываем в него значения
// из переменной Spis_Rez
Spis_Rez->SaveToFile(Rez_f);
ShowMessage("Файл с итоговыми результатами готов!");
}/// PyskClick

void __fastcall TYdal_Lish_Prob::VuxodClick(TObject *Sender)
{
    delete Spis_Rez;
    delete Spis_Isx;
    Close();
}///VuxodClick
```

Пользовательская функция *Vvod_Spiska* записывает из файла *Isx_f.txt* исходные строки текста в переменную *Spisok_Isx* типа *TStringList**. При этом в фактическом параметре *i_n* функции *Vvod_Spiska*, передаваемом в функцию по ссылке (как параметр-переменная), фиксируется *фактическое* число строк исходного текста.

Далее в переменную *Spisok_Rez* типа *TStringList** записывается поясняющая надпись: *Исходный текст с лишними пробелами:*. При помощи функции *Vvod_Spiska* ниже этой надписи записываются строки исходного текста. Ниже указанного текста в переменной *Spisok_Rez* записывается пустая строка и заголовок: *Текст после обработки:*.

Функция *Ydal_Prob* при помощи цикла *for* последовательно просматривает все строки исходного текста (строки переменной *Spisok_Isx*). В ходе такого просмотра в *каждой* строке вначале устраняются *все* начальные и конечные пробелы, а затем в начало и конец строки добавляется по одному пробелу. Далее при помощи цикла *while* и функции *Pos* находится позиция двух пробелов, после чего один из них уничтожается применением функции *Delete*, затем вновь функция *Pos* находит очередные два пробела и вновь один из них уничтожается функцией *Delete*. Так происходит до тех пор, пока все непрерывные последовательности пробелов длиной в два и более пробелов не уменьшатся до *одного* пробела. После этого функция *Pos* возвращает нуль, что позволяет циклу *while* завершить свою работу на выбранной строке.

Цикл *for* последовательно запускает внутренний цикл *while* для каждой строки исходного текста – от первой до последней (*i_n*) и очищает тем самым весь текст от лишних пробелов.

Результаты преобразования исходного текста помещаются в переменной *Spisok_Rez* при помощи *повторного* вызова пользовательской функции *Vvod_Spiska*. Стандартная функция *SaveToFile* текст из переменной *Spisok_Rez* выводит в файл *Rez_f.txt*. Таким образом, в файле *Rez_f.txt* находятся исходный и преобразованный тексты с комментариями к ним.

В этом приложении используются *две* переменные типа *TStringList**. В одну из них загружен исходный текст из файла *Isx_f.txt*, а в другую помещались комментарии, исходный текст и текст после обработки. В конце работы программы содержимое переменной *Spisok_Rez* записывалось в файл *Rez_f.txt*. Эту же задачу решим теперь с использованием только *одной* переменной *Spis* типа *TStringList**.

В неё запишем вначале исходный текст, а затем – всё, что ранее помещалось в переменную *Spis_Rez*. Для реализации этой цели применим массив строк типа *String*, в который перепишем исходный текст из переменной *Spis*. Затем, очистив переменную *Spis*, внесём в неё все комментарии, первоначальный и итоговый текст, после чего её содержимое занесём в файл *Rez_f.txt*.

Для обработки многострочных текстов массивы строк привлекается очень часто, поэтому весьма полезно познакомиться с кодом программы, в котором используется этот тип данных. В этом варианте программы иллюстрируется использование функции *AnsiPos()* вместо метода *Pos()* класса *String*.

Код второго варианта приложения *Удаление лишних пробелов*

```
const int nn= 100;//Максимальная длина массива
//Тип int_Mas: массивы с числом компонент nn типа int
typedef String S_Mas[nn];//Определение пользовательского типа
TStringList *Spis= new TStringList;
String Isx_f, Rez_f;//Файловые переменные

void Vvod(S_Mas & SMas, int & iSt)
{
    try
    {
        Spis->LoadFromFile(Isx_f);
    }//try
    catch (...)
    {
        ShowMessage("Файл \" " + Isx_f + " \" не найден");
        Abort();
    }//catch
    iSt= Spis->Count;//Определяем число строк в списке
    for (int ij= 0; ij< iSt; ++ij)//Заносим текст в массив
    {
        Trim(Spis->Strings[ij]);
        Spis->Strings[ij]= " " + Spis->Strings[ij] + " ";
        SMas[ij]= Spis->Strings[ij];
    }//for_ij
    Spis->Clear();//Очистка переменной Spis
}//Vvod

void Vuvod(S_Mas SMas, int iSt)
{//Данные из Spis копируются в SMas
    for (int ij= 0; ij< iSt; ++ij)
        Spis->Add(SMas[ij]);
}//Vuvod

void Ydal_Prob(S_Mas & SMas, int iSt)
{
    for (int ij= 0; ij< iSt; ++ij)
    {
        //int ii= SMas[ij].Pos(" ");//Вариант кода
        int ii= AnsiPos(" ",SMas[ij]);//Вариант кода
        while (ii> 0)
        {
            SMas[ij].Delete(ii,1);//Уничтожили один пробел
```

```

        // Положение следующих двух пробелов
        ii= SMas[ij].Pos("  ");
    }//while
} //for_ij
} //Ydal_Prob

void __fastcall TLish_Prob_Mas::PyskClick(TObject *Sender)
{
    int i_n; //Фактическое количество строк в исходном тексте
    S_Mas St_Mas; //Объявление переменной типа S_Mas
    Isx_f= Isx_Fajl->Text; //Ввод имён файлов
    Rez_f= Rez_Fajl->Text;
    Vvod(St_Mas, i_n);
    //Добавляем новую строку списка
    Spis->Add("Исходный текст с лишними пробелами");
    Vuvod(St_Mas, i_n);
    Spis->Add(""); //Пропуск строки
    Spis->Add("Текст после обработки:");
    Ydal_Prob(St_Mas, i_n);
    Vuvod(St_Mas, i_n);
    //В файл Rez_f записываем данные из переменной Spis
    Spis->SaveToFile(Rez_f);
    ShowMessage("Файл с итоговыми результатами готов!");
} // PyskClick

void __fastcall TLish_Prob_Mas::VuxodClick(TObject *Sender)
{
    delete Spis;
    Close();
} //VuxodClick

```

Программа *Поиск и замена*

У любого текстового редактора, в том числе и во всех средах программирования высокого уровня, имеется функция поиска и замены фрагмента текста. Реализуем эту задачу в учебном приложении *Поиск и замена*, при этом проиллюстрируем применение отдельных функций, рассмотренных на этом уроке.

Программа загрузит текст из выбранного файла, и все вхождения в него заданного замещаемого фрагмента заменит текстом замещающего фрагмента, сообщит число сделанных замен, а изменённый текст запишет в файл, имя которого укажет пользователь приложения. Интерфейс программы показан на рис. 12.4. В этом приложении применяются те же пользовательские функции *Vvod_Spiska*, *Vuvod_Spiska* и *VuxodClick*, которые использовались в первом варианте приложения *Удаление лишних пробелов*. Поэтому в коде настоящего приложения указаны только их заголовки. Объекты типа *TEdit* интерфейса, предназначенные для ввода заменяемого и замещающего фрагментов, имеют соответственно имена *Poisk* и *Zamena*.

Код приложения *Поиск и замена*

```

TStringList *Spis_Isx= new TStringList,
                  *Spis_Rez= new TStringList;
String Isx_f, Rez_f;

```

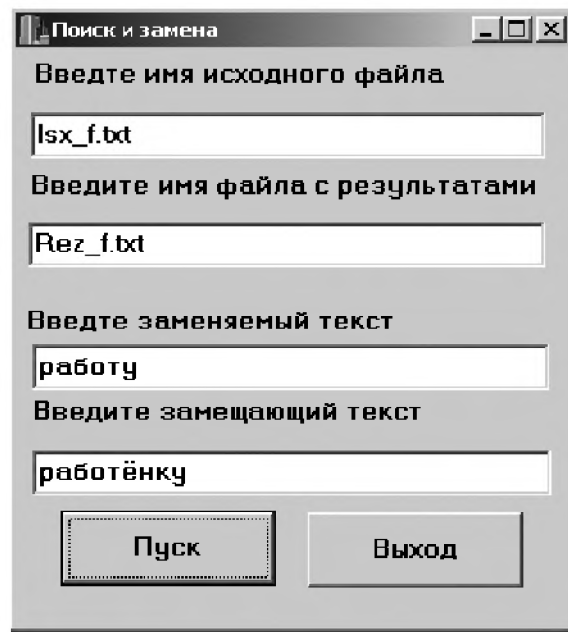


Рис. 12.4. Интерфейс приложения Поиск и замена

```

void Vvod_Spiska(int & iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов

void Vuvod_Spiska(int iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов

void Poisk_Zamena(int iSt, String sPoisk, String sZamena)
{
    int Ch_Zamen = 0; //Число замен
    for (int ij= 0; ij< iSt; ++ij)
    {
        int ii= Spis_Isx->Strings[ij].Pos(sPoisk);
        while (ii> 0)
        {
            //Удаляем слово sPoisk с позиции ii
            Spis_Isx->Strings[ij]= Spis_Isx->Strings[ij].
                Delete(ii,sPoisk.Length());
            //Вставляем слово sZamena
            Spis_Isx->Strings[ij]= Spis_Isx->Strings[ij].
                Insert(sZamena,ii);

            Ch_Zamen++;
            // Положение нового вхождения слова sPoisk
            ii= Spis_Isx->Strings[ij].Pos(sPoisk);
        } //while
    } //for_ij
    ShowMessage("Сделано " + IntToStr(Ch_Zamen) + " замен");
} //Poisk_Zamena

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    int i_n; //Фактическое число строк исходного текста
    Isx_f= Isx_Fajl->Text; //Ввод имён файлов
    Rez_f= Rez_Fajl->Text;
    String s_Poisk= Poisk->Text,

```

```
//s_Zamena= Trim(Zamena->Text);
s_Zamena= Zamena->Text;
Vvod_Spiska(i_n);
//Добавляем новую строку списка
Spis_Rez->Add("Исходный текст:");
Vuvod_Spiska(i_n);
Spis_Rez->Add(""); //Пропуск строки
Spis_Rez->Add("Текст после обработки");
Poisk_Zamena(i_n, s_Poisk, s_Zamena);
Vuvod_Spiska(i_n);
//В файл Rez_f записываем данные из переменной Spis_Rez
Spis_Rez->SaveToFile(Rez_f);
ShowMessage("Файл с итоговыми результатами готов!");
} // PyskClick

void __fastcall TForm1::VuxodClick(TObject *Sender)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов
```

Это приложение прекрасно справляется с возложенными на него задачами. Однако в случае очень длинных строк работа пользовательской функции *Poisk_Zamena* окажется не очень эффективной. Ведь после замены очередного вхождения искомого фрагмента текста цикл осуществляет поиск нового вхождения фрагмента текста с начала строки, а не с позиции, расположенной после заменённого фрагмента. Таким образом, программа *впустую* просматривает часть строки, где замены уже сделаны, и поэтому требуемый фрагмент текста там искать бессмысленно.

В предлагаемом варианте функции *Poisk_Zamena* поиск заданного фрагмента текста методом *Pos* осуществляется *только* в *оставшейся части* строки. Это оказывается возможным с использованием метода *SubString*, который вырезает часть строки от позиции *i_Pred* за вставленным фрагментом вплоть до конца строки. Новое число символов в строке после вставки очередного замещающего фрагмента определяется методом *Length* и записывается в переменную *Dlina_Stroki*. Скорость поиска заменяемого фрагмента несколько повысится, если каждый раз не вычислять новую длину строки *Dlina_Stroki*, а вместо этой переменной в функции *SubString* указать, например, константу 255 (или 2555) – ведь длина строки всегда будет меньше такого значения. В этом случае функция *SubString* также вырежет часть строки от указанной позиции до её *фактического* конца.

Код варианта функции *Poisk_Zamena*

```
void Poisk_Zamena(int iSt, String sPoisk, String sZamena)
{
    int Ch_Zamen = 0;
    for (int ij= 0; ij< iSt; ++ij)
    {
        // Индекс, предшествующий первому символу
        int i_Pred= 0, //ещё необработанной строки
            ii= Spis_Isx->Strings[ij].Pos(sPoisk);
        while (ii> i_Pred)
        {
```

```

//Удаляем слово sPoisk с позиции ii
Spis_Isx->Strings[ij]= Spis_Isx->Strings[ij].
                        Delete(ii, sPoisk.Length());
//Вставляем слово sZamena
Spis_Isx->Strings[ij]= Spis_Isx->Strings[ij].
                        Insert(sZamena, ii);
Ch_Zamen++;
//Номер позиции символа за вставленным словом
i_Pred= ii + sZamena.Length();
int Dlina_Stroki= Spis_Isx->Strings[ij].Length();
// Позиция нового вхождения слова sPoisk
//в оставшемся тексте
ii= i_Pred -1 +
    //Позиция искомого слова в части строки
    //после вставленного слова
    Spis_Isx->Strings[ij].
        SubString(i_Pred, 255).Pos(sPoisk);
    SubString(i_Pred, Dlina_Stroki).Pos(sPoisk);
} //while
} //for_ij
ShowMessage("Сделано " + IntToStr(Ch_Zamen) + " замен");
} //Poisk_Zamena

```

Реализовать эту же пользовательскую функцию вполне возможно без использования методов *Delete* и *Insert*. После определения методом *Pos* позиции *ii* начала искомого фрагмента в выбранной строке, следует вырезать два фрагмента этой строки методом *SubString*. Первый фрагмент – это часть строки от её начала до позиции *ii* – 1. Второй фрагмент включает в себя часть строки от позиции за исключаемым фрагментом текста до конца строки. Между указанными частями текста методом конкатенации вставляется замещающий фрагмент текста *sZamena*. Поиск последующих вхождений искомого фрагмента осуществляется только в ещё не откорректированной части строки.

Код ещё одного варианта функции *Poisk_Zamena*

```

void Poisk_Zamena(int iSt, String sPoisk, String sZamena)
{
    int Ch_Zamen = 0;
    for (int ij= 0; ij< iSt; ++ij)
    {
        // Индекс, предшествующий первому символу
        int i_Pred= 0, //ещё не обработанной части строки
        ii= Spis_Isx->Strings[ij].Pos(sPoisk);
        while (ii)
        {
            Spis_Isx->Strings[ij]=
                //Часть строки до найденного слова
                Spis_Isx->Strings[ij].
                    SubString(1, ii + i_Pred -1) +
                sZamena + //Вставляем слово sZamena
                //Добавляем часть строки, расположенную
                //после уничтоженного слова
                Spis_Isx->Strings[ij].
                    SubString(ii + i_Pred + sPoisk.Length(), 255);

```

```
Ch_Zamen++;
//Номер позиции символа за вставленным словом
i_Pred += ii - 1 + sZamena.Length();
// Позиция нового вхождения слова sPoisk
//в оставшемся тексте
ii= Spis_Isx->Strings[ij].
        SubString(i_Pred + 1, 255).Pos(sPoisk);
    }//while
}//for_ij
ShowMessage("Сделано " + IntToStr(Ch_Zamen) + " замен");
}//Poisk Zamena
```

Самый краткий вариант пользовательской функции *Poisk_Zamena* получается при использовании функции *StringReplace*. В этом случае следует лишь каждую строку текста обрабатывать такой функцией, никакие хитрые алгоритмы не нужны. Незначительный недостаток применения функции *StringReplace* заключается лишь в том, что невозможно определить число сделанных ею замен. В этом варианте предусмотрена обработка ситуации, когда искомый фрагмент не будет найден ни в одной из строк исходного текста. Ранее такую возможность не обрабатывали лишь в целях компактности кода (что в методическом плане очень плохо).

Самый краткий вариант функции *Poisk_Zamena*

```
void Poisk_Zamena(int iSt, String sPoisk, String sZamena)
{
    int iChislo= 0;//Счётчик строк без искомого фрагмента
    for (int ij= 0; ij< iSt; ++ij)
    {
        String Stroka= Spis_Isx->Strings[ij];
        Spis_Isx->Strings[ij]=
            StringReplace(Spis_Isx->Strings[ij], sPoisk, sZamena,
                          TReplaceFlags()<<rfReplaceAll);
        if (Stroka!= Spis_Isx->Strings[ij])
            iChislo++;
    }//for_ij
    if (iChislo== 0)//Строки не изменились
        ShowMessage("Текст не найден");
}//Poisk_Zamena
```

Вопросы для самоконтроля

- ☐ Как программно установить код выбранной литеры?
- ☐ Какие *Esc*-последовательности осуществляют переход на следующую строку и вставку знака табуляции в выводимом тексте?
- ☐ Укажите способ вывода одинарных и двойных кавычек в тексте сообщений.
- ☐ Когда используются символы *ANSI* и *OEM*?
- ☐ С какой целью в конец текстовой переменной типа массив *char* записывается нулевой символ? Укажите начальный индекс таких массивов.
- ☐ Чем отличаются переменные типа указатель на *char* и массив *char*?

- ☐ Разрешается ли в ходе выполнения программы изменять длину инициализированной строки типа массив символов *char*?
- ☐ Можно ли в переменную типа *String* в ходе выполнения программы записывать строки разной длины? Каким значением инициализируется такая переменная по умолчанию?
- ☐ В каком случае текстовую переменную нельзя преобразовать в число?
- ☐ Сколько символов можно записать в строку типа *String*? Каким индексом начинается нумерация литер в такой строке?
- ☐ Назовите минимальный и максимальный объёмы переменной типа *String*? Чем определяется её минимальный объём?

12.4. Задачи для программирования

Задача 12.1. В файле *Isx_f.txt* запишите текст, состоящий из нескольких строк. Программа должна в файл *Rez_f.txt* записать исходный текст, текст после шифровки и дешифровки с соответствующими комментариями. Метод шифровки определите самостоятельно.

Задача 12.2. В файл *Isx_f.txt* запишите текст, состоящий из нескольких строк. В файл *Rez_f.txt* программа должна записать с соответствующими комментариями исходный текст и слово текста с максимальным числом гласных букв.

Задача 12.3. В файл *Isx_f.txt* запишите до десяти предложений, каждое предложение расположите на отдельной строке. В файл *Rez_f.txt* программа должна записать исходный текст и текст после обработки. Обработка заключается в том, что, если в двух соседних предложениях имеются одинаковые слова, то в первом из них все символы одинаковых слов следует заменить цифрами, совпадающими с номером этой строки.

12.5. Варианты решений задач

Код решения задачи 12.1

```
//Интерфейс приложения такой же, как в приложении Удаление
//лишних пробелов (кроме заголовка)
TStringList *Spis_Isx= new TStringList,
               *Spis_Rez= new TStringList;
String Isx_f, Rez_f;

void Vvod_Spiska(int & iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов

void Vuvod_Spiska(int iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов

void Shifr_DeShifr(int iSt, int Shifr)
{
```



```
String Stroka;
for (int ij= 0; ij< iSt; ++ij)
{
    Stroka= Spis_Isx->Strings[ij];
    for (int ii= 1; ii<= Stroka.Length(); ++ii)
        Stroka[ii]-= Shifr;
    Spis_Isx->Strings[ij]= Stroka;
} //for_ij
} //Shifr_DeShifr

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    int i_n; //Фактическое число строк исходного текста
    Isx_f= Isx_Fajl->Text; //Ввод имён файлов
    Rez_f= Rez_Fajl->Text;
    Vvod_Spiska(i_n);
    //Добавляем новую строку списка
    Spis_Rez->Add("Исходный текст:");
    Vuvod_Spiska(i_n);
    Spis_Rez->Add(""); //Пропуск строки
    Spis_Rez->Add("Текст после шифровки:");
    Shifr_DeShifr(i_n, 3); //Уменьшаем код каждого символа на 3
    Vuvod_Spiska(i_n);
    Spis_Rez->Add(""); //Пропуск строки
    Spis_Rez->Add("Текст после дешифровки");
    Shifr_DeShifr(i_n, -3); //Возвращаем прежний код символу
    Vuvod_Spiska(i_n);
    Spis_Rez->SaveToFile(Rez_f);
    ShowMessage("Файл с итоговыми результатами готов!");
} // PyskClick

void __fastcall TForm1::VuxodClick(TObject *Sender)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов
```

Код решения задачи 12.2

```
//Интерфейс приложения такой же, как в приложении Удаление
//лишних пробелов (кроме заголовка)
TStringList *Spis_Isx= new TStringList,
               *Spis_Rez= new TStringList;
String Isx_f, Rez_f,
//Слово с максимальное количество гласных букв
    Max_Glas_Slovo;
int iMax; //Максимальное количество гласных букв

void Vvod_Spiska(int & iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов

void Vuvod_Spiska(int iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов

void Ydal_Prob(int iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов
```

```

void Max_Glas(int iSt)//Определяет слово с максимальным
                    //количеством гласных букв
{
    String Stroka,
        Glasnue= "AaEeИиOoУуЫыЮюЯя",
        Slovo;
    for (int ij= 0, ip= 0; ij< iSt; ++ij)
    {
        Stroka= Spis_Isx->Strings[ij];
        if (Stroka.IsEmpty())//Пропуск пустых строк
            continue;
        Stroka= Trim(Stroka);//Убрали начальные и конечные пробелы строки
        //Текст строки должен начинаться и заканчиваться
        Stroka= " " + Stroka + " "; // одним пробелом
        //Позиция первого пробела в строке
        //ip= AnsiPos(" ", Stroka);//Вариант кода
        ip= Stroka.Pos(" ");
        while(ip> 0 && ip<= Stroka.Length())
        {
            int ik= 1;//Номер буквы в слове
            while(Stroka[ip+ik]!=' ')
                ik++;//Счётчик числа букв в слове
            //Выделяем слово
            Slovo= Stroka.SubString(ip+1, ik-1); //ik Поскольку
            //пробел в конце слова не записывается в Slovo
            int Kol_Glas= 0;
            for (int ii= 1; ii<= Slovo.Length(); ++ii)
                if (Glasnue.Pos(Slovo[ii])> 0)
                    //if (AnsiPos(Slovo[ii], Glasnue)> 0)//Вариант кода
                    //if (IsDelimiter(Glasnue, Slovo, ii))//Вариант
                    Kol_Glas++;//Счётчик числа гласных букв
            if (ij== 0 && ip== 1)
            {
                //Число гласных в первом слове текста
                iMax= Kol_Glas;
                Max_Glas_Slovo= Slovo;
            }//if_1
            if (Kol_Glas> iMax)
            {
                iMax= Kol_Glas;
                Max_Glas_Slovo= Slovo;//Текущее слово с
                //максимальным количеством гласных
            }//if_2
            //Для продвижения к следующему пробелу
            Stroka[ip]= '*';
            //Выделяем необработанную часть строки
            //Stroka= Stroka.SubString(ip+ik, 255);//Вариант
            //Позиция пробела перед следующим словом
            ip= Stroka.Pos(" ");
            //ip= AnsiPos(" ", Stroka);//Вариант
            //Чтобы не выйти за границу строки
            if (ip== Stroka.Length())
                ip= 0;
        }//while_ip
    }//for_ij
}

```

```

} //Max_Glas

//Этот оригинальный вариант функции Max_Glas() предложен
//четырнадцатилетней Олей Копыловой весной 2005 года
void Poisk(String Stroka, String &IskSlovo, int &MaxKolGl)
{
    int Posic = 1,
        KolGl = 0;
    String Slovo;
    for (int ij = 1; ij <= Stroka.Length(); ++ij)
    {
        if(IsDelimiter("аеёиоуыэюяАЕЁИОУЫЭЮЯ", Stroka, ij))
        {
            ++KolGl;
        } //if_IsDelim
        if(IsDelimiter(" ", Stroka, ij))
        {
            Slovo = Stroka.SubString(Posic, ij + 1 - Posic);
            if(KolGl > MaxKolGl)
            {
                IskSlovo = Slovo;
                MaxKolGl = KolGl;
            } //if_KolGl
            KolGl = 0;
            Posic = ij;
        } //if_Stroka
    } //for
} //Poisk

//К варианту Копыловой
void Max_Glas(int iSt) //Определяет слово с максимальным
                        //количеством гласных букв
{
    for(int ii = 0; ii < iSt; ++ii)
        Poisk(Spis_Isx->Strings[ii], Max_Glas_Slovo, iMax);
} //Max_Glas

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    int i_n; //Фактическое число строк исходного текста
    Isx_f = Isx_Fajl->Text; //Ввод имён файлов
    Rez_f = Rez_Fajl->Text;
    Vvod_Spiska(i_n);
    Spis_Rez->Add("Исходный текст:");
    Vuvod_Spiska(i_n);
    Spis_Rez->Add(""); //Пропуск строки
    Ydal_Prob(i_n);
    Max_Glas(i_n);
    // \n- перевод строки
    Spis_Rez->Add("Слово с максимальным числом гласных\
букв: \n" + Max_Glas_Slovo + " содержит " +
        IntToStr(iMax) + " гласных");
    Spis_Rez->SaveToFile(Rez_f);
    ShowMessage("Файл с итоговыми результатами готов!");
} // PyskClick

void __fastcall TForm1::VuxodClick(TObject *Sender)

```

```
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов
```

Решение задачи 12.3

```
//Интерфейс приложения такой же, как в приложении Удаление
//лишних пробелов (кроме заголовка)
```

```
TStringList *Spis_Isx= new TStringList,
             *Spis_Rez= new TStringList;
String Isx_f, Rez_f;
```

```
void Vvod_Spiska(int & iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов
```

```
void Vuvod_Spiska(int iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов
```

```
void Ydal_Prob(int iSt)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов
```

```
void ZamenaSlov(int iSt)//Одинаковые слова
//смежных строк в верхней строке заменяются цифрами,
// совпадающими с номером строки
{
    String Stroka_1, Stroka_2, Slovo, Slovo_n,
    Tekst_Vid_Nom_Str;
    int ip1, ip2;//Число букв в слове и позиция его
    //начала в Stroka_1 и пробела перед ним в Stroka_2
    for (int ij= 0; ij< iSt - 1; ++ij)
    {
        if (Spis_Isx->Strings[ij].IsEmpty())
            continue;
        //Преобразуем число в текст
        Tekst_Vid_Nom_Str= IntToStr(ij + 1);
        Stroka_1= Spis_Isx->Strings[ij];
        Stroka_2= Spis_Isx->Strings[ij + 1];
        Stroka_1= Trim(Stroka_1);//Убрали начальные и
        Stroka_2= Trim(Stroka_2);//конечные пробелы строк
        //Тексты строк должны начинаться и заканчиваться
        Stroka_1= " " + Stroka_1 + " "; // пробелом
        Stroka_2= " " + Stroka_2 + " ";
        //Первая позиция первого пробела в Stroka_2
        ip2= Stroka_2.Pos(" ");
        // или ip2= AnsiPos(" ", Stroka_2);
        while(ip2> 0 && ip2<= Stroka_2.Length())
        {
            int ik= 1;//Номер буквы в слове
            while(Stroka_2[ip2 + ik]!=' ' &&
                (ip2 + ik)<= Stroka_2.Length())
            //Нет выхода за границу Stroka_2
            //Счётчик числа букв и последнего пробела
            ik++; // в слове
            //Выделяем слово в Stroka_2
            Slovo= Stroka_2.SubString(ip2 + 1,ik - 1);
```

```

//Позиция Slovo в Stroka_1
ip1= Stroka_1.Pos(Slovo);
//или ip1= AnsiPos(Slovo, Stroka_1);
//Анализируем Slovo, а не буквосочетание
if (ip1> 0 &&
    Stroka_1[ip1 + Slovo.Length()]== ' '
    && Stroka_1[ip1 - 1]== ' ')
{
    while(ip1> 0 &&
        Stroka_1[ip1 + Slovo.Length()]== ' '
        && Stroka_1[ip1 - 1]== ' ')
    {
        Stroka_1= Stroka_1.
            Delete(ip1, Slovo.Length());
        Slovo_n= "";
        for (int ii= 1; ii<= Slovo.Length(); ii++)
            Slovo_n= Slovo_n + Tekst_Vid_Nom_Str;
        //Набор символов, количество которых
        //равно длине слова
        Stroka_1= Stroka_1.Insert(Slovo_n, ip1);
        //Следующая позиция Slovo в Stroka_1
        ip1= Stroka_1.Pos(Slovo);
    }//while_ip1
    //Для продвижения по словам Stroka_2
    Stroka_2[ip2]= '*';
    //Следующая позиция ' ' в Stroka_2
    ip2= Stroka_2.Pos(" ");
    //или ip2= AnsiPos(" ", Stroka_2);
    if (ip2== Stroka_2.Length())
        ip2= 0;
    }//if
else
{
    //Для продвижения по словам Stroka_2
    Stroka_2[ip2]= '*';
    //Следующая позиция ' ' в Stroka_2
    ip2= Stroka_2.Pos(" ");
    //ip2= AnsiPos(" ", Stroka_2);
    if (ip2== Stroka_2.Length())
        ip2= 0;
    }//else
    }//while_ip2
    Spis_Isx->Strings[ij]= Stroka_1;
}//for_ij
}//ZamenaSlov

```

Этот вариант функции *ZamenaSlov()* предложен четырнадцатилетней Олей Копыловой весной 2005 года.

```

void Zamena(TStringList *Spis, int iSt)
{
    String Slovo;
    for(int ii = 0; ii < iSt - 1; ++ii)
        for(int ij = 1, Posic = 1; ij <=
            Spis->Strings[ii+1].Length(); ++ij)

```


```

{
    if(IsDelimiter(" .,:\"'!\", Spis->Strings[ii+1], ij))
    {
        Slovo = Spis->Strings[ii+1].
            SubString(Posic, ij - Posic);
        Posic = ij+1;
        String NovStr;
        for(int in = 0; in < Slovo.Length(); ++in)
            NovStr = NovStr + IntToStr(ii + 1);
        Spis->Strings[ii] =
            StringReplace(Spis->Strings[ii], Slovo, NovStr,
                TReplaceFlags()<<rfReplaceAll<<rfIgnoreCase);
    } //if(Stroka)
} //for
} //Zamena

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    int i_n; //Фактическое число строк исходного текста
    Isx_f= Isx_Fajl->Text; //Ввод имён файлов
    Rez_f= Rez_Fajl->Text;
    Vvod_Spiska(i_n);
    Spis_Rez->Add("Исходный текст:");
    Vuvod_Spiska(i_n);
    Spis_Rez->Add(""); //Пропуск строки
    Spis_Rez->Add("Текст после обработки:");
    Ydal_Prob(i_n);
    ZamenaSlov(i_n); //Вариант автора
    //Zamena(Spis_Isx, i_n); //Вариант Оли Копыловой
    Vuvod_Spiska(i_n);
    Spis_Rez->SaveToFile(Rez_f);
    ShowMessage("Файл с результатами готов!");
} //PyskClick

void __fastcall TForm1::VuxodClick(TObject *Sender)
//Тело функции см. в первом варианте приложения
//Удаление лишних пробелов

```



Урок 13. Перечисления, множества, объединения и структуры

13.1. Перечислимый тип переменных

С первого занятия *настоятельно* рекомендовалось числовые константы в программе заменять их *именованными* константами (строковыми или символьными), поскольку в этом случае код приложения становится более ясным, легко масштабируемым, удобным для последующей модернизации. Если серия именованных констант участвует в сходных операциях, то разумно объединить их в группу. Такой тип данных называется *перечислением* (а элементы группы именуется константами перечисления), он широко используется в стандартных функциях *Builder*. Например, режимы работы функции *FileOpen*, предназначенной для открытия двоичного файла (см. девятый урок), обозначены ясными константами-перечислениями *fmOpenRead*, *fmOpenWrite*, *fmOpenReadWrite*. При этом для *каждой* из этих констант имеется числовой эквивалент в шестнадцатеричной системе счисления.

Использование перечислений является хорошим стилем программирования, делает код приложения не только легко понятным, но и более надёжным. Ведь переменной этого типа можно назначить *только* те значения, которые заранее *перечислены* при её объявлении. У переменных перечислимого типа имеются те же полезные ограничения, что и у именованных констант:

- ❑ Имена констант-перечислений должны быть *уникальными*: не повторяться внутри отдельного перечисления (внутри одной группы) и в разных перечислениях (в разных группах).
- ❑ После объявления значение константы-перечисления *нельзя изменить* (оно не может стоять *слева* оператора присваивания).
- ❑ При выводе на экран (на печать) выводится не строковое (символьное) имя константы-перечисления, а её *числовое* значение.

Программист по своему усмотрению может определить тип переменной-перечисления, в котором *перечислить* весь набор требуемых значений этой переменной. Вот пример объявления и инициализации таких переменных:

```
enum Givotnue {Kot, Sobaka, Loshadj, Korova} Givotn, Domashn_Giv;
```

Тип перечисления с именем *Givotnue* вводится с использованием служебного слова *enum* (*enumerate* – перечислять), в фигурных скобках, через запятую пере-

числяются строковые (обычно) константы – все значения перечисления. После закрывающей фигурной скобки через запятую указываются идентификаторы переменных введенного типа (в примере *Givotn, Domashn_Giv*). Введенное имя типа перечисления (в нашем примере это *Givotnue*), при необходимости может использоваться далее в программе для ввода *новых* переменных этого же типа:

```
Givotnue Svinka, Petyx;
```

Объём переменной типа *enum* составляет один байт.

В ходе компиляции всем строковым константным значениям перечисления по умолчанию присваиваются их номера во введенном списке, нумерация списка начинается с *нуля*. Однако *любому* значению перечисления *при* его *объявлении* можно присвоить *произвольный* номер с положительным или отрицательным *целочисленным* значением, например *Sobaka*= 54. Если же какой-либо константе-перечислению путём присваивания не задан определённый номер, то он оказывается на единицу *больше* номера предшествующей константы в списке. В частности в результате присваивания *Sobaka*= 54 окажется, что *Loshadj*==55, *Korova*==56.

В справочной системы *Builder* сообщается, что по умолчанию переменным перечислимого типа присваиваются значения констант, с которых начинается список перечислений. В нашем примере должно быть *Givotn*== *Kot*, *Domashn_Giv*== *Kot*. Однако на практике это правило *практически никогда* не выполняется, поэтому не рекомендуем его использовать. Введенным переменным, например, *Givotn* и *Domashn_Giv* в программе разрешается назначать *любые допустимые* значения как в числовом, так в строковом виде (только из заявленного списка констант).

Перечисления принадлежат к порядковому типу данных, поэтому *общее* число значений перечисления *ограничено*, каждому строковому значению соответствует определённое число. По этой причине перечисления допустимо использовать в операторах циклов (например, *for*) и *switch*. Числа (иногда их называют номерами), присвоенные строковым константам-перечислениям, разрешается описывать *любым* целочисленным типом данных. *Наибольшее* значение такой константы составляет 4 294 967 295, этому же числу соответствует *наибольшее* количество констант в перечислении. Вместе с тем согласно справочной системе, диапазон значений констант простирается от – 2 147 483 648 до 2 147 483 647, что *на одну* константу меньше упомянутого максимального количества констант. Константам-перечислениям при их объявлении разрешается присваивать *совпадающие* значения.

Имя типа конкретного перечисления и имена переменных введенного типа перечисления разрешается *не вводить*, если в программе они не используются. Ниже приводится поясняющий пример:

```
//Нет имён типа и переменных
enum {Kot, Sobaka, Loshadj, Korova};
for ( int ii= Kot; ii<= Korova; ii++)
{
    switch (ii)
    {
```



```

    case Kot: ShowMessage("Кот");
        break;
    case Sobaka: ShowMessage("Собака");
        break;
    case Loshadj: ShowMessage("Лошадь");
        break;
    case Korova: ShowMessage("Корова");
} //switch
} //for

```

Переменные-перечисления и сами константы-перечисления могут участвовать в операциях сравнения и служить операндами *любых* допустимых математических выражений:

```

enum Givotnue {Kot, Sobaka, Loshadj, Korova} Givotn, Giv;
Giv= Sobaka; //1
Givotn= 2;
if (Giv< Givotn)
    ShowMessage(Giv + Givotn); //1 + 2== 3

```

13.2. Множества

Этот тип данных отсутствует в *C++*. Однако его применение в *TurboPascal*, а затем и *Delphi* показало, как изящно с его использованием решаются многие задачи. При этом сокращается размер программ, их код становится более ясным. Поэтому включение множеств в *Builder* очень правильный шаг разработчиков изучаемой среды программирования.

Множество – это набор однотипных элементов порядкового типа данных, количество элементов во множестве не превышает 256. *Минимальное* значение элемента множества не может быть отрицательным числом, а *максимальное* – не больше 255. Вот так с использованием служебных слов *typedef* и *Set* (помещать) объявляется тип *Latin_Bykvi* множества, в котором могут размещаться *только* прописные и строчные латинские буквы.

```
typedef Set <char, 'A', 'z'> Latin_Bykvi;
```

В угловых скобках указаны тип элементов множества (*char*), его начальный минимальный ('A') и конечный максимальный ('z') элементы. Напоминаем, что строчная буква имеет больший номер в таблице кодов, чем прописная (подробнее см. раздел 12.1).

При помощи введенного имени типа *Latin_Bykvi* объявим *две* переменные типа множества с идентификаторами *Da* и *Net*:

```
Latin_Bykvi Da, Net;
```

Эти же переменные можно объявить и так:

```
Set <char, 'A', 'z'> Da, Net;
```

Упомянутыми способами можно объявить *произвольное* число переменных-множеств с другими именами и *все* они будут *однотипными*: их можно сравнивать, объединять и производить над ними другие операции, сущность которых разъясняется в п.13.2.1.

Порождённые переменные-множества *Da* и *Net* пока *пусты*, в них нет *ни од-*

ного элемента. Для помещения во множество *допустимых* элементов (тех, что перечислялись при объявлении типа) существует специальная операция: *помещение во множество* «<<». Применим её для помещения во множество *Da* букв *Y* и *y*, а во множество *Net* – букв *N* и *n*:

```
Da<< 'Y'<< 'y';
Net<< 'N'<< 'n';
```

При объявлении множеств на основе типа *char* имеет смысл использовать *только* символы первой половины таблицы кодов, которые принадлежат стандартным символам *ASCII*. Недопустимо применять при объявлении множества *одновременно* (совместно) символы латиницы и кириллицы. Символы кириллицы, в принципе, *могут* участвовать в объявлении множеств: на стадии компиляции программы нет сообщения об ошибке. Однако после процедуры *наполнения* порождённой переменной-множества буквами кириллицы такое множество по-прежнему останется пустым:

```
typedef Set <char, 'A', 'я'> Kirillica; //А- буква кириллицы
Kirillica Kiril, Pystoe;
Kiril<< 'Ю'<< 'я';
if (Pystoe== Kiril)
    ShowMessage("Множества пусты!");
```

На экран выводится сообщение: *Множества пусты!* Таким образом, символы из второй части таблицы кодов *не заносятся* во множество. Здесь иллюстрируется также проверка однотипных множеств на равенство. Множества *равны*, если они пусты, либо все их элементы тождественны. Порядок помещения элементов во множество *не имеет значения*, вводимые элементы множества *могут повторяться* – это не влияет на его состав и величину. Вместе с тем рекомендуется всё же помещать элементы во множество в каком-то порядке, поскольку это облегчает знакомство с его составом.

Всякая *set*-переменная хранит не сами значения из базового типа (перечисленные при его объявлении), а лишь *бинарную* информацию о каждом таком значении – есть оно во множестве или его там нет. Более подробно механизм реализации этих особенностей разъяснён в п. 13.2.2. Размер типа *Set* и объём множественных переменных как пустых, так и наполненных равен 8 байтам (объём не зависит от числа помещённых в них допустимых элементов множества).

Применение множественных переменных покажем в приложении, на форме которого имеется объект *Label1* с надписью: *Продолжить работу программы?* Ниже этого заголовка находится поле ввода объекта *Edit1*, предназначенное для ввода информации (ответ пользователя относительно продолжения работы программы). При вводе литеры *Y* (в любом регистре) следует сделать так, чтобы появилось сообщение: *Работайте дальше*, а при занесении строчной либо прописной буквы *N*, необходимо завершить работу приложения. Строку кода, предназначенную для предварительной очистки поля ввода (*Edit1->Text= ""*;) можно разместить в конструкторе приложения, либо произвести очистку в *Инспекторе Объектов* в ходе проектирования приложения.

Как известно, у объектов формы и у самой формы имеется набор различных событий, которые могут происходить в ходе выполнения приложения. Познакомимся с очередными двумя такими событиями. Событие *OnKeyPress* наступает при нажатии *клавиши символа*, когда фокус активности находится в поле ввода (в нашем примере объекта *Edit1*). Это событие происходит *перед* отображением символа нажатой клавиши в поле ввода. Необходимо особо отметить, что при этом *распознаются* символы кириллицы и латиницы, знаки пунктуации, цифры и другие символы в нижнем и верхнем регистре. Однако упомянутое событие *не наступает* для функциональных клавиш и кнопок мыши. Вместе с тем *распознавание* нажатия указанных элементов клавиатуры и мыши осуществляется событием *OnKeyDown*, которое в то же время не распознаёт нажатие клавиш символов, названных при описании события *OnKeyPress*.

В заготовку функции *Edit1KeyPress* по обработке события *OnKeyPress* включим код по реализации требуемых реакций приложения:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, char &Key)
{
    typedef Set <char, 'A', 'z'> Latin_Bykvu;
    Latin_Bykvu Da, Net;
    Da<< 'Y'<< 'y'; //Включение допустимых элементов во множество
    Net<< 'N'<< 'n';
    if (Da.Contains(Key))
        ShowMessage("Работайте дальше");
    else
        if (Net.Contains(Key))
            Close();
} //Edit1KeyPress
```

В *Builder* тип множество реализован как шаблон класса, поэтому функции по обработке переменных этого типа данных являются методами экземпляра этого класса, и они вызываются с использованием операции точка и имени объекта (экземпляра класса). У множества имеется всего *два* метода. Метод *Contains(Argyment)* возвращает переменную логического типа *bool* и предназначен для проверки наличия во множестве заданного элемента *Argyment*: *Da.Contains(Key)*. В функции *Edit1KeyPress* аргумент *Key* является параметром-переменной.

Метод *Clear()* очищает множество, исключает из него все элементы, в результате чего оно (в нашем последующем примере *Da*) становится пустым: *Da.Clear()*.

Приведём ещё одну иллюстрацию применения множества. На форме приложения (с именем *Form1*) имеется всего лишь три объекта: объект *Edit1* – для *ввода* чисел; объект *Label1* – для *вывода* указания: *Введите число*; кнопка с именем *Button1* и заголовком *Пуск*, после её нажатия выводится сообщение о значении введенного числа. При нажатии на клавишу нецифрового символа (*Key*) раздаётся короткий звуковой сигнал, а сам символ *не заносится* в поле ввода. Ниже показан конструктор и другие функции этого приложения.

```
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
    Edit1->Text= ""; //Начальная очистка поля ввода
```

```

Label1->Caption= "Введите число";
} // TForm1

void __fastcall TForm1::Edit1KeyPress(TObject *Sender, char &Key)
{
    Set <char, '0', '9'> Cifru;
    Cifru<< '0'<< '1'<< '2'<< '3'<< '4'<< '5'<< '6'<< '7'<< '8'<< '9';
    if (!Cifru.Contains(Key)) // Нажата нецифровая клавиша
    {
        Key= NULL; // стираем введенную литеру
        Beep(); // Звуковой сигнал
    } // if
} // Edit1KeyPress

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    String Stroka= Edit1->Text;
    ShowMessage("Введено " + Stroka);
    Edit1->Text= ""; // Очистка поля ввода
} // Button1Click

```

13.2.1. Операции над однотипными множествами

Какие либо совместные операции с двумя множествами могут производиться *только*, когда они *однотипны*. Для множеств разных типов *никакие* совместные операции не предусмотрены.

Объединение двух множеств *Set_A* и *Set_B* осуществляется при помощи знака «+»: *Set_A + Set_B*. В результате порождается *новое* множество, которое состоит из элементов, принадлежащих *одновременно* двум упомянутым множествам. Приведём поясняющий пример.

```

typedef Set <char, 'A', 'z'> Lat_Byk;
Lat_Byk Da_Y, Da_y, Da_Y_y;
Da_Y<< 'Y';
Da_y<< 'y';
Da_Y_y= Da_Y + Da_y; // Объединение множеств
if (Da_Y_y.Contains('Y') && Da_Y_y.Contains('y'))
    ShowMessage("Множества объединены!");

```

На экране появится сообщение, свидетельствующее о том, что в новом множестве присутствуют оба элемента из объединённых множеств.

К множествам применима также операция «+=», при её использовании в предшествующем примере можно не вводить переменную-множество *Da_Y_y*:

```

Da_Y+= Da_y; // Объединение множеств
if (Da_Y.Contains('Y') && Da_Y.Contains('y'))
    ShowMessage("Множества объединены!");

```

Пересечение множеств *Set_A* и *Set_B*, записываемое как *Set_A*Set_B*, порождает новое множество, состоящее *только* из тех элементов, которые принадлежат как множеству *Set_A*, так и множеству *Set_B*. Если в исходных множествах нет общих элементов, то результатом пересечения будет пустое множество.

```

Lat_Byk Lena, Misha, Peresechenie, Pystoe, Ky_Ky;
Lena<< 'L'<< 'e'<< 'n'<< 'a';

```

```

Misha<< 'M'<< 'i'<< 's'<< 'a';
Peresechenie= Lena*Misha;//Пересечение множеств
if (Peresechenie.Contains('a'))
    ShowMessage("Есть такая буква!");
Ky_Ky<< 'Y'<< 'u'<< 'P';
Peresechenie.Clear();//Очистка
Peresechenie= Lena*Ky_Ky;
if (Peresechenie==Pystoe )
    ShowMessage("Множества пусты!");

```

Поскольку в пересекаемых множествах *Lena* и *Misha* имеется общая буква *a*, то вначале в результате работы вышеприведенного фрагмента появится сообщение: *Есть такая буква!* Затем очищенное множество *Peresechenie* по-прежнему останется пустым, поскольку в пересечении *Lena*Ky_Ky* нет общих букв, что и подтвердит сообщение: *Множества пусты!*

Результат первой части этого примера достигается также при помощи операции «*=»:

```

Lena*= Misha;
if (Lena.Contains('a'))
    ShowMessage("Есть такая буква!");

```

Во множестве *Lena* будет находиться только буква *a*, поскольку путём присваивания результата пересечений множеств *Lena* и *Misha* (буквы *a*) предшествующее содержимое множества *Lena* заменяется присваиваемым значением.

Разность множеств *Set_A* и *Set_B* определяется такой записью *Set_A – Set_B*. В результате разности упомянутых множеств появляется *новое* множество, в котором имеются только те элементы множества *Set_A*, которые *отсутствуют* во множестве *Set_B*: из множества *Set_A* *удаляются* все элементы, обнаруженные во множестве *Set_B* (общие элементы первоначальных множеств). Согласно такому правилу множество *Set_A – Set_B* и множество *Set_B – Set_A* не равны друг другу.

```

Lat_Byk L_M, M_L;
L_M= Lena - Misha;//Разность множеств
M_L= Misha - Lena;//Разность множеств
if (L_M!= M_L)
    ShowMessage("Разные множества!");
Lena-= Misha;//Разность множеств, иллюстрация операции «-=»
if (!Lena.Contains('a'))
    ShowMessage("Нет такой буквы!");

```

На экран последовательно выводятся сообщения *Разные множества!* и *Нет такой буквы!*

В ходе работы программы в уже существующие переменные-множества операциями «<<» и «>>» можно добавлять и удалять допустимые элементы. *Builder* не сообщит об ошибке, если из множества удаляется отсутствующий в нём элемент. Приведём поясняющие примеры.

```

Lena.Clear(), Misha.Clear();
Lena<< 'L'<< 'e'<< 'n'<< 'a';
Misha<< 'M'<< 'i'<< 's'<< 'a';
Lena>> 'L'>> 'e'>> 'n';
Misha>> 'M'>>'i'>> 's'>>'h';
if (Lena==Misha)

```

```
ShowMessage("Множества равны!");
Misha>>'T';//Не является ошибкой
```

Присваивание множества:

```
Lena.Clear(), Misha.Clear();
Misha<< 'M'<< 'i'<< 's'<< 'a';
Lena= Misha;//Lena== Misha
```

Сравнение множеств.

Два множества *Set_A* и *Set_B* равны друг другу, если каждый элемент из *Set_A* является также элементом *Set_B* и наоборот. В этом случае *Set_A == Set_B*. Если это условие не выполняется, то множества не равны друг другу: *Set_A != Set_B*. В *Builder* других операций сравнения множеств нет.

Старшинство множественных операторов.

Наивысшим приоритетом обладает пересечение, меньший уровень старшинства у объединения, разности, добавления << и исключения >> элементов. Низший приоритет выполнения операций принадлежит операциям сравнения: ==, != и методу *Contains(Argument)*. Метод *Contains()* позволяет выяснить имеется ли заданный элемент у выбранного множества. Примеры использования упомянутых операций показаны ниже.

```
Lat_Byk Lena, Misha, Peres;
Lena<< 'L'<< 'e'<< 'n'<< 'a';
Misha<< 'M'<< 'i'<< 's'<< 'a';
Peres= Lena*Misha;//Множество Peres состоит только из 'a'
if (Lena - Lena*Misha << 'a' == Lena && Peres.Contains('a'))
    ShowMessage("Покажусь на экране!");
```

На экране появится сообщение: *Покажусь на экране!* Правила приоритета операций совпадают с приоритетом похожих арифметических операций.

13.2.2. Внутреннее представление множеств

В конечном итоге *все* возможные значения *заданного* множества представляются в памяти компьютера последовательностями битов одинаковой длины. Число ячеек (битов) в этой последовательности определяется разностью максимального и минимального элементов. За *каждое* значение множества отвечает *отдельный* бит. Если множество *содержит* некоторый элемент, то в выделенной для него битовой ячейке памяти записывается 1, если не содержит, то в неё помещается 0. Приведём поясняющий пример. Пусть задано множество *Set_X*:

```
Set <int, 1, 5> Set_X;//00000
```

В комментарии приведена последовательность из пяти нулей, поскольку множество *Set_X*, в котором могут храниться целые числа от 1 до 5, является ещё пустым и поэтому все имеющиеся в нём ячейки заполнены нулями.

```
Set_X<< 5<< 4; //00011
```

После добавления в это множество чисел 5 и 4 на местах в бинарной последовательности, в которых диагностируются наличие или отсутствие заданных эле-

ментов множества, появились единицы, указывающие на то, что во множество *Set_X* помещены числа 4 и 5.

Теперь в это же множество добавим остальные допустимые элементы – числа 1, 2 и 3:

```
Set_X << 1 << 2 << 3;    //11111
```

Во множество *занесены* все допустимые элементы, поэтому на всех позициях его внутреннего представления находятся *только* единицы.

Метод *Contains()* определяет, находится ли требуемый элемент во множестве или нет: он выясняет номер этого элемента во внутреннем представлении множества и проверяет, что находится в ячейке с этим номером. Если там находится 0, то элемент во множестве отсутствует, если же там 1 – он там имеется.

Рассмотренные выше операции над множествами сводятся к *поразрядным логическим операциям* над последовательностью битов. *Логическое сложение* или *поразрядная дизъюнкция* « \vee » выполняется так: $0 \vee 0 = 0$; $0 \vee 1 = 1$; $1 \vee 0 = 1$; $1 \vee 1 = 1$. В информационных технологиях её ещё называют *поразрядным ИЛИ* и используют знак « $|$ ». В *Builder* для её осуществления дополнительно имеется *своя* поразрядная логическая операция *or*. При помощи поразрядного *ИЛИ* (логического сложения) битов реализуется объединение множеств:

```
Set <int, 1, 5> Set_X, //00000
                Set_Y, //00000
                Set_Z; //00000
```

```
Set_X << 2 << 3;    //01100
Set_Y << 3 << 4;    //00110
Set_Z = Set_X + Set_Y; //01110
```

В результате объединения множеств *Set_X* и *Set_Y* порождается множество, в котором находятся числа 2, 3 и 4. Поэтому во второй, третьей и четвёртых ячейках (битах) записаны единицы. Следует отметить, что поразрядные операции входят в набор команд процессора, поэтому выполняются очень быстро.

Логическое умножение или *поразрядная конъюнкция* « \wedge » описывается такими действиями: $0 \wedge 0 = 0$; $0 \wedge 1 = 0$; $1 \wedge 0 = 0$; $1 \wedge 1 = 1$. В информатике эта операция называется *поразрядным И*, для неё используют знак « $\&$ ». *Builder* для осуществления этих же действий предлагает дополнительно свою поразрядную логическую операцию *and*. Эта поразрядная операция позволяет реализовать пересечение прежних множеств *Set_X* и *Set_Y*:

```
Set_Z = Set_X * Set_Y;    //00100
```

Единица присутствует только на третьей позиции в последовательности битов, поскольку во множествах *Set_X* и *Set_Y* число 3 является общим элементом.

Теперь покажем, как с использованием поразрядных операций осуществляется *разность* двух множеств. С этой целью напомним внутреннюю структуру использованных выше множеств *Set_X* и *Set_Y* и приведём содержимое *инвертированного* множества *Set_Y*:

```
Set_X << 2 << 3;    //01100
Set_Y << 3 << 4;    //00110
Инвертированное Set_Y; //11001
```

Далее осуществим операцию *and* (поразрядное *И*) для множества *Set_X* и инвер-

тированного *Set_Y*:

```
Set_X           //01100
Инвертированное Set_Y //11001
Set_Z = Set_X - Set_Y; //01000
```

Как видите, указанные поразрядные операции позволили осуществить разность двух множеств. После исключения из множества *Set_X* общего с множеством *Set_Y* элемента «3» в новом множестве *Set_Z* находится только элемент «2», вследствие этого единица оправданно заняла свою позицию во второй ячейке битовой реализации порождённого множества *Set_Z*. Таким образом, разность множеств *Set_X* и *Set_Y* осуществляется поразрядной операцией *and* над множеством *Set_X* и инвертированным множеством *Set_Y*.

Ранее отмечалось, что множество *Set_X - Set_Y* и множество *Set_Y - Set_X* это разные множества.

```
Set_Z = Set_Y - Set_X; //00010
```

При таком вычитании множеств из множества *Set_Y* исключается общий с множеством *Set_X* элемент «3», поэтому в новом множестве *Set_Z* записывается только «4», что отмечается единицей в четвёртой позиции битовой реализации этого множества. Теперь покажем, как этот же результат достигается при помощи упомянутых поразрядных операций:

```
Set_Y           //00110
Инвертированное Set_X //10011
Set_Z = Set_Y - Set_X; //00010
```

Приведенная выше поразрядная арифметика легко реализуется электронными схемами и является основой работы ЭВМ.

13.2.3. Пример применения множеств

В заключение настоящего раздела изучите фрагмент приложения, в котором применяются как перечисления, так и множества. Имеется группа из 10 студентов, каждый из которых посещает одну или две спортивные секции. Требуется выявить тех учащихся, которые посещают секции каратэ и бокса, карате и шейпинга. Последовательность шагов при решении этой задачи может быть такой. Объявим переменную-перечисление с именем *Gryppa* и типом *Stydentu*. В типе *Stydentu* перечислим имена студентов (записанные латиницей) в произвольном порядке. Заметим, что в программе имя упомянутой переменной и имя её типа не используются. Далее объявим множества с именами *Karate*, *Boks*, *Shejping*, которые затем наполним различными числами, соответствующими номерам имён студентов, перечисленных при объявлении перечисления.

Цикл *for* последовательно перебирает все имена из группы студентов. Для удобства счёта первому студенту группы присвоен номер, равный единице. По умолчанию он должен быть равным нулю. При помощи оператора *switch* имена студентов, записанные латиницей, преобразуем в имена, выводимые кириллицей, с применением метода *Contains()* решим вопрос о выполнении упомянутых условий для каждого члена студенческой группы.


```
typedef enum Stydentu {Lena= 1, Misha, Olga, Vladimir, Yuriy,
                      Sergej, Viktor, Natasha, Masha, Sasha} Gryppa;
Set <int, 1, 10> Karate, Boks, Shejping;
Karate << 2 /*Misha*/<< 4/*Vladimir*/<< 5/*Yuriy*/
      << 7/*Viktor*/<<6/*Sergej*/<< 1/*Lena*/<< 3/*Olga*/;
Boks << 6/*Sergej*/<< 7/*Viktor*/ <<10 /*Sasha*/;
Shejping << 1/*Lena*/<< 3/*Olga*/<< 8 /*Natasha*/<< 9/*Masha*/;
String Stroka, Stroka_1, Stroka_2;
for (int ii= Lena; ii<= Sasha; ii++)
{
    switch (ii)
    {
        case Lena:Stroka= "Лена"; //1
            break;
        case Misha:Stroka= "Миша";//2
            break;
        case Olga:Stroka= "Ольга";//3
            break;
        case Vladimir:Stroka= "Владимир";//4
            break;
        case Yuriy:Stroka= "Юрий";//5
            break;
        case Sergej:Stroka= "Сергей";//6
            break;
        case Viktor:Stroka= "Виктор";//7
            break;
        case Natasha:Stroka= "Наташа";//8
            break;
        case Masha:Stroka= "Маша";//9
            break;
        case Sasha:Stroka= "Саша";//10
    }
    //switch
    if (Karate.Contains(ii) && Boks.Contains(ii))
        Stroka_1= Stroka_1 + " " + Stroka;
    if (Karate.Contains(ii) && Shejping.Contains(ii))
        Stroka_2= Stroka_2 + " " + Stroka;
}
//for_ii
ShowMessage("Секцию каратэ и бокса посещают: " + Stroka_1);
ShowMessage("Секцию каратэ и шейпинга посещают: " + Stroka_2);
```

В *Builder* объём множественной переменной определяется количеством элементов множества, заявленных при объявлении типа. Так, если число элементов менее 7, то объём переменной составляет 1 байт; от 8 до 15 – 2 байта, от 16 до 23 – 4 байта, далее происходит увеличение объёма на один байт при увеличении числа элементов множества на 7 элементов.

13.3. Объединения

Ранее применялись переменные, в которых в ходе работы программы можно было хранить лишь *один* тип данных, заданный при их объявлении. Объединения – это тип переменных, позволяющий в одной и той же области памяти помещать (объединять) *разнотипные* данные (конечно, в несовпадающие моменты време-

ни). Такое устройство переменных-объединений позволяет сэкономить оперативную память, её объём определяется не суммарным объёмом всех переменных объединения, а лишь объёмом *максимальной* из них.

Приведём пример объявления типа объединения с именем *TUnion*:

```
const int n= 6;
union TUnion
{
    int Chislo_int;
    double Chislo_double;
    char * Yk_char;
    char Bykva;
    int Mas_int[n];
    enum {Vchera= 11, Zavtra};
}; //TUnion
```

Следует заметить, что среди типов переменных, объединённых в таком типе данных, *не может* быть тип *String* (поскольку объединения разрабатывались в то время, когда тип *String* ещё не был предложен). Для объявления *union*-переменных, например, с именами *x* и *y*, поступают традиционным образом:

```
TUnion x, y;
```

Переменные можно ввести сразу же после объявления (описания) типа:

```
union TUnion
{
    ... //Ввод полей объединения
} x, y; //TUnion
```

Использование *union*-переменной показано в следующем фрагменте программы, в котором эта переменная *последовательно* инициализируется значениями *разных* типов. Просмотр введенных значений осуществляется функцией *ShowMessage()*.

```
for (int ii= 1; ii<= n; ii++)
{
    switch (ii)
    {
        case 1: x.Chislo_int= 1949;
                ShowMessage(x.Chislo_int);
                break;
        case 2: x.Chislo_double= 54.636363;
                ShowMessage(x.Chislo_double);
                break;
        case 3: x.Yk_char= "Привет!";
                ShowMessage(x.Yk_char);
                break;
        case 4: x.Bykva= 'Ю';
                ShowMessage(x.Bykva);
                break;
        case 5: x.Mas_int[4]= 12;
                ShowMessage(x.Mas_int[4]);
                break;
        case 6: ShowMessage(x.Vchera + x.Zavtra); //11+12==23
    } //switch
}
```

```
}//for_ii
```

На экране последовательно появятся сообщения: 1949; 54.636363; *Привет!*; Ю; 12; 23 (11 + 12 = 23).

Все переменные, описанные в объединении *x*, имеют единый адрес, поскольку блок ячеек для каждой переменной начинается с одной и той же ячейки, однако число ячеек (байт) у разных переменных может быть различным. В подтверждение этого приведём сравнение адресов лишь двух переменных объединения:

```
if (&x.Mas_int==&x.Bykva)
    ShowMessage("Адреса совпадают!");
```

На экране появится сообщение: *Адреса совпадают!* Объединения часто применяются в структурах, назначение и использование которых детально описано в следующем разделе.

13.4. Структуры

13.4.1. Простые структуры

Объекты материального и духовного мира отличаются друг от друга набором характеристик. Например, конкретный товар имеет розничную цену, дату изготовления, габаритные размеры, вес, цвет, гарантийный срок, фирму-изготовителя и другие параметры. Приведенные характеристики определяются числами *разных* типов и строками-названиями. Многие параметры товара имеют *свой* набор характеристик. Например, у компьютеров процессоры отличаются тактовой частотой, типом, гарантийным сроком службы, напряжением питания и другими показателями. Структуры *предназначены* для того, чтобы *сохранять и обрабатывать* столь разноликие данные.

Массивы также позволяют хранить данные структурированного типа, однако компоненты массивов могут быть *только одного* типа. Структуры же имеют возможность хранить данные *разных* типов и содержать внутри себя ещё и функции, предназначенные для обработки своих же данных. Покажем вначале, как объявляются структуры, в которых нет внутренних функций.

Объявление структурного типа данных проиллюстрируем описанием отдельных индивидуальных характеристик сотрудника государственного учреждения или частной фирмы.

```
struct strt_Sotr// Задание имени типа
{
    //Поля для хранения фамилии, имени и отчества сотрудника
    String Fam, Im, Otch;
    unsigned long Ind;//Почтовый индекс
    String Gorod,//Город проживания
        Yl,//Улица
        Nom_Dom;//Номер дома
    short Nom_Kv;//Номер квартиры
    String Dolgn;//Должность
    float Okl;//Оклад
};//strt_Sotr
```

За служебным словом *struct* ставится имя типа структуры, после которого в фигурных скобках располагаются переменные-характеристики абстрактного сотрудника. Такие переменные называют полями. Впереди имени каждого поля указывается его тип, однотипные поля разрешается перечислять через запятую. В конце определения полей и за закрывающей скобкой структуры ставится точка с запятой. Все имена полей должны быть уникальными – запрещены одноимённые поля внутри одной и той же структуры.

Объявлением структуры *конкретного* типа среде программирования *лишь* сообщается, что в программе переменные такого типа теперь *могут* (имеют право) создаваться, однако до момента порождения они *не существуют*. Аналогичная ситуация наблюдается и при объявлении пользовательского типа (с использованием служебного слова *typedef*). Казалось бы – это достаточно очевидное заключение, однако, мы считаем, не лишним его подчеркнуть. Следует всегда также помнить, что при объявлении структурного типа *запрещается* инициализировать поля.

Для *создания* переменных структурного типа используется стандартная операция, применяемая для объявления переменных *произвольного* типа. Вот так она выглядит при порождении переменной типа *strt_Sotr* с именем *Sotr*:

```
strt_Sotr Sotr;
```

Переменную (или ряд переменных) можно задать сразу же после объявления типа:

```
struct strt_Sotr
{
    ...//Описание полей структуры
} Slygaschij, Kollega;//strt_Sotr
```

Выше проиллюстрированы способы задания переменных структурного типа, которые используются в *C++*. В *C++Builder* *дополнительно* к этим вариантам разрешается при объявлении переменных структурного типа впереди имени типа указывать ещё и служебное слово *struct*:

```
struct strt_Sotr Nashi_Sotr, Moi_Sotr;
```

Нам представляется это излишним, поэтому далее такой вариант задания переменных не используется.

При объявлении переменных структурного типа выделяется *только место* для размещения полей переменной, например, с именем *Sotr*. В самих же полях находятся пустые строки, если их тип *String*, либо неприсвоенные значения, если они числового типа (включая массивы) или типов *char* и *char**. Блоки ячеек памяти, выделяемые для размещения полей структуры, располагаются *в той же последовательности*, в какой они объявлены в структуре. Один блок следует за другим вплотную друг к другу. Поэтому все блоки в совокупности составляют *единый* блок оперативной памяти, выделяемый для размещения *всей* структуры. Доступ к выбранному полю осуществляется при помощи имени переменной-структуры (например, *Sotr*) и имени поля, которые разделяются точкой (без пробелов слева и справа от точки). Вот так выполняется присваивание допустимых значений полям переменной *Sotr*:

```
Sotr.Fam= "Иваненко";
Sotr.Im= "Иван";
Sotr.Otch= "Иванович";
Sotr.Ind= 62461;
Sotr.Gorod= "Южный";
Sotr.Yl= "Артёма";
Sotr.Nom_Dom= "45/2";
Sotr.Nom_Kv= 54;
Sotr.Dolgn= "старший научный сотрудник";
Sotr.Okl= 813.6;
```

Теперь содержимое *любого* поля можно показать на экране, например при помощи функции *ShowMessage*:

```
ShowMessage(FloatToStrF(Sotr.Okl, ffFixed, 8, 2));
//Появится на экране 813.60;
ShowMessage(Sotr.Fam + " " + Sotr.Im +
" " +Sotr.Otch); //Иваненко Иван Иванович
```

На структурный тип, точно так же как и на любой другой тип переменных, можно объявить переменную-указатель, и направить его на ранее порождённую переменную-структуру:

```
strt_Sotr *Yk;
Yk= &Sotr;
```

Теперь, для вывода на экран прежней информации можно использовать имя указателя *Yk*, а для доступа к его полям требуется применять операцию стрелка «*→*» (а, не операцию точка «*.*»):

```
ShowMessage(FloatToStrF(Yk->Okl, ffFixed, 8, 2));
ShowMessage(Yk->Fam + " " + Yk->Im + " " + Yk->Otch);
```

Существует *ещё один* способ доступа к элементам указателя на структуру: с использованием операции разыменования.

```
ShowMessage(FloatToStrF((*Yk).Okl, ffFixed, 8, 2));
```

Обратите внимание на круглые скобки вокруг имени указателя: (**Yk*). Операция разыменования «***» должна относиться *только* к имени указателя, а не к имени поля переменной структурного типа.

Полями структуры могут быть простые типы данных (*int*, *float* и др.), массивы, перечисления, объединения, пользовательские типы и переменные-структуры *других* типов. Вместе с тем структуре не разрешается включать поле, тип которого *совпадает* с типом самой структуры, однако позволены поля-указатели на одноимённый тип структуры:

```
struct strt_Sotr
{
    str_Sotr Rabotnik; //Недопустимый тип поля
    strt_Sotr *Rabotnik; //Допустимый тип поля
}; //strt_Sotr
```

После приведенных разъяснений будет более ясным следующее определение структуры – это структурированный тип данных и функций их обработки, который состоит из *фиксированного* числа полей-компонентов *разного* типа. Разрешены структуры и с однотипными полями. Функции структур называются компонентными функциями или методами (о них рассказывает п. 13.4.4).

13.4.2. Вложенные структуры

При описании сложных объектов используют структуры, в которых в качестве полей применяются другие структуры. Структура с разветвлёнными полями напоминает устройство матрёшки. *Максимальное* число вложений (или глубина) определяется с учётом суммарного объёма всех других переменных стека или оперативной памяти (следовательно, зависит от места расположения структурной переменной). Правила работы с вложенными структурами и преимущества, которые достигаются при их применении, покажем на прежней структуре, преобразовав её во вложенную структуру *strt_Sotr*. Вначале требуется описать *менее* сложные структуры.

```
struct strt_Adres
{
    unsigned long Ind;
    String Gorod, Yl, Nom_Dom;
    short Nom_Kv;
}; //strt_Adres

struct strt_Rabota
{
    String Dolgn;
    float Okl;
}; //strt_Rabota
```

Затем объявляются те структуры, в которых применяются приведенные выше более простые структуры.

```
struct strt_Sotr
{
    String Fam, Im, Otch;
    strt_Adres Adres;
    strt_Rabota Rabota;
} Sotr;
```

Ещё раз отмечаем, что *вначале* описаны две простые структуры *strt_Adres* и *strt_Rabota*, в их полях соответственно содержится информация относительно адреса местожительства сотрудника и его занятости (работы). *Затем*, в структуре *strt_Sotr* помимо простых полей типа *String*, предназначенных для размещения фамилии, имени и отчества сотрудника, имеются два поля *Adres* и *Rabota*, тип которых задаётся упомянутыми выше более простыми структурами.

Здесь иллюстрируется процесс упорядочивания *однородных* данных по структурам с ясными именами. Этот процесс аналогичен упорядочению файлов по соответствующим каталогам и подкаталогам. Так же как имена файлов в каталогах, имена полей вложенных структур могут повторяться (совпадать, быть одноимёнными) *только* на *разных* уровнях вложенности. Однако имеются и *существенные* отличия. Вначале создаётся каталог, а затем – его подкаталоги. При определении же вложенных структур вначале описываются структуры *самого нижнего* уровня вложенности, и лишь затем – менее глубокие структуры и так далее. Последней описывается структура, вобравшая в себя все предшествующие

структуры. Такой порядок согласуется с правилом *всех* языков программирования высокого уровня – вначале опиши (переменную, тип и др.), а затем уж используй.

Теперь, после объявления переменной-структуры (любым способом) с именем *Sotr*, доступ к её сложному полю осуществляется с использованием имени соответствующей более простой структуры.

```
Sotr.Fam= "Иваненко";  
Sotr.Im= "Иван";  
Sotr.Otch= "Иванович";  
Sotr.Adres.Ind= 62461;  
Sotr.Adres.Gorod= "Южный";  
Sotr.Adres.Yl= "Артёма";  
Sotr.Adres.Nom_Dom= "45/2";  
Sotr.Adres.Nom_Kv= 54;  
Sotr.Rabota.Dolgn= "старший научный сотрудник";  
Sotr.Rabota.Okl= 813.6;  
ShowMessage(Sotr.Fam);  
ShowMessage(Sotr.Adres.Gorod);  
ShowMessage(FloatToStrF(Sotr.Rabota.Okl, ffFixed, 8, 2));
```

На экране появятся сообщения: *Иваненко; Южный; 813.60*.

13.4.3. Массивы структур

В предшествующем пункте рассмотрена только одна переменная *Sotr* типа *strt_Sotr*. Однако чаще приходится сталкиваться с одномерным массивом, каждый элемент которого имеет тип *strt_Sotr*.

Пусть, например, имеется малое предприятие с десятью сотрудниками, индивидуальные данные каждого из них определяется структурой типа *strt_Sotr*, вся же такая база данных является массивом *Mas_Sotr[n]*, где *n* – число сотрудников. Ниже покажем, как определяется такой массив данных, и проиллюстрируем доступ к отдельным полям структуры для четвёртого элемента массива (пятого сотрудника).

```
const int n= 10; //Число сотрудников  
strt_Sotr Mas_Sotr[n]; //Задание массива из n элементов  
Mas_Sotr[4].Im= "Пётр";  
Mas_Sotr[4].Rabota.Dolgn= "мастер цеха";  
  
ShowMessage(Mas_Sotr[4].Im);  
ShowMessage(Mas_Sotr[4].Adres.Gorod);  
ShowMessage(Mas_Sotr[4].Rabota.Dolgn);
```

На экран последовательно выводится: *Петр; Харьков; мастер цеха*.

Имя переменной и все поля отделяются друг от друга точкой, имена полей повышающихся уровней вложенности располагаются справа налево. Как видите, перед именем поля имеется путь к нему: имя переменной и имена полей всех уровней вложенности. В результате наличия таких префиксов имена переменных программы *могут совпадать* с именами полей структур *любой* вложенности: в ходе компиляции программы *Builder* по составному имени отличает поле структуры от идентификатора переменной.

Можно *дополнительно* ввести массив:

```
strt_Sotr Rabotniki[n];
```

В этом случае массивы *Rabotniki* и *Mas_Sotr* *однотипны*, и к ним можно применить, например, такое присваивание:

```
Rabotniki[2]= Mas_Sotr[4];
```

Теперь *все* поля структуры, расположенной в ячейке массива *Rabotniki*[2], заменены *соответствующими* полями структуры, помещёнными в ячейке *Mas_Sotr*[4]. Таким образом, в упомянутых ячейках *разных* массивов находятся *одни и те же* данные.

13.4.4. Структуры с вариантными полями

В структуру *strt_Sotr* включим вариантное поле с именем *Tel_EMail*. Это поле имеет тип *unsigned long*, если сотрудник указал для связи с ним контактный телефон, и тип *char ** в случае, когда связь осуществляется при помощи электронной почты *E-Mail*. Чтобы программа могла определить тип данных этого поля для *конкретного* сотрудника (это важно как для чтения, так и записи информации), включим в структуру ещё одно поле с типом *char* и именем *Kom* (коммутатор). Буква *t* в этом поле указывает на то, что в поле *Tel_EMail* помещён номер телефона, буква *e* – электронный адрес.

```
//Используем объединение для создания вариантного поля
```

```
union Var_Un
```

```
{
    unsigned long Tel;
    char * E_Mail;
}; //Var_Un
```

```
struct strt_Sotr
```

```
{
    String Fam, Im, Otch;
    strt_Adres Adres ;
    strt_Rabota Rabota;
    char Kom;//t, e
    Var_Un Tel_EMail;
} Mas_Sotr[n]; //strt_Sotr
```

```
Mas_Sotr[4].Kom= 't';
```

```
if (Mas_Sotr[4].Kom== 't')
```

```
    Mas_Sotr[4].Tel_EMail.Tel= 380509772182;
```

```
ShowMessage(Mas_Sotr[4].Tel_EMail.Tel); // 380509772182
```

```
Mas_Sotr[4].Kom= 'e';
```

```
if (Mas_Sotr[4].Kom== 'e')
```

```
    Mas_Sotr[4].Tel_EMail.E_Mail= "Yuriy.P.Fedorenko@Univer.Kharkov.ua";
```

```
ShowMessage(Mas_Sotr[4].Tel_EMail.E_Mail);
```

```
//Yuriy.P.Fedorenko@Univer.Kharkov.ua
```

Как использовать вариантное поле структуры при обработке базы данных, расположенной в текстовом файле, показано в п. 13.4.6.

13.4.5. Методы структур

Поля структур можно обрабатывать при помощи стандартных и пользовательских функций (как обычные переменные). Однако такой *свободный* доступ к полям *понижает* надёжность программ, поскольку в этом случае для обработки поля может быть ошибочно вызвана не та функция, которая предназначена для его обработки. Поэтому в структурах предусмотрено, чтобы функции для обработки их полей описывались *внутри* самой же структуры. Такие функции иногда называют компонентными функциями, однако более часто их именуют *методами*. Обрабатывать поля внешними функциями не рекомендуется.

Объединение в структурах (и классах, см. след. урок) данных и функций для их обработки называется *инкапсуляцией*. Инкапсуляция позволяет осуществить *наследование* полей и методов вложенных структур. Вложенные структуры называют *родительскими*, а структуры, их поглотившие, – *потомками* или *дочерними* структурами. Применение механизма наследования структур *повышает* надёжность программ и *ускоряет* их модернизацию. Посмотрите, как задача п. 13.4.2 выглядит в случае описания методов структуры *внутри* пользовательской функции (например, *PyskClick*):

```
struct strt_Sotr
{
    String Fam, Im, Otch;
    strt_Adres Adres ;
    strt_Rabota Rabota;
    void Pokaz_Oklad(void)
    {
        ShowMessage(FloatToStrF(Rabota.Okl, ffFixed, 8, 2));
    } //Pokaz_Oklad

    void Pokaz_FIO(void)
    {
        ShowMessage(Fam + " " + Im + " " + Otch);
    } //Pokaz_FIO
} Sotr;

Sotr.Fam= "Иваненко";
Sotr.Im= "Иван";
Sotr.Otch= "Иванович";
Sotr.Rabota.Okl= 813.6;
Sotr.Pokaz_Oklad(); //813.60
Sotr.Pokaz_FIO(); //Иваненко Иван Иванович
```

Рекомендуем структуру описывать *вне* пользовательской функции, например, в файле реализации или в заголовочном файле (в классе формы). Последний вариант предпочтительнее, однако, для удобства чтения *настоящего* пособия объявим структуру в файле реализации. При объявлении структуры вне пользовательской функции методы структуры внутри самой структуры *можно* не описывать, а ограничиться лишь указанием их прототипа. При этом код методов структуры *обязательно* должен приводиться *вне* пользовательской функции.

```
struct strt_Sotr
{
```

```
String Fam, Im, Otch;
strt_Adres Adres ;
strt_Rabota Rabota;
void Pokaz_Oklad(void); //Прототип
void Pokaz_FIO(void); //Прототип
} Sotr;

void strt_Sotr::Pokaz_Oklad(void)
{
    ShowMessage(FloatToStrF(Rabota.Okl, ffFixed, 8, 2));
} //Pokaz_Oklad//;

void strt_Sotr::Pokaz_FIO(void)
{
    ShowMessage(Fam + " " + Im + " " + Otch);
} //Pokaz_FIO;
```

В пользовательской функции (например, *PyskClick*) остались операторы присваивания значений полей и вызовы методов структуры *strt_Sotr*.

Структуру можно также описывать в отдельном заголовочном файле. Описание же методов структуры лучше оставить в файле реализации. В этом случае облегчается модернизация приложения: в реализацию методов можно вносить изменения, полностью заменять их тела без каких-либо изменений описания самой структуры.

В показанном способе объявления структуры описание методов (вне структуры) осуществляется с применением имени структуры и операции разрешения области действия «::»: *void strt_Sotr::Pokaz_Oklad(void)*. В данном примере упомянутая операция указывает на то, что метод *Pokaz_Oklad(void)* принадлежит структуре с типом *strt_Sotr*. При этом следует отметить, что в *каждом* экземпляре структуры (или переменной типа *struct*) хранятся *только* значения её полей, методы же конкретной структуры существуют в *единственном* числе и *располагаются в типе*, при необходимости они лишь вызываются внутрь переменной-структуры для обработки её полей. Следует помнить, что методы структуры *не копируются* в её экземпляры.

13.4.6. Пример обработки базы данных

Учебные программы с использованием структур достаточно *громоздки*. С этим фактом необходимо *смириться* и не смущаться объёмных листингов последующих приложений. На самом деле в них *нет* ничего сложного. Проиллюстрируем возможности структур на примере обработки упрощённой учебной базы данных.

Такая база данных имеет вид таблицы и расположена в текстовом файле *Baza.txt*. Информация о *каждом* сотруднике учреждения приведена в отдельной строке файла, а однородные данные для *всех* сотрудников оформлены в виде столбцов: фамилия; имя; отчество; почтовый индекс местожительства; названия города и улицы; номера дома и квартиры; должность; оклад; в последнем столбце указывается номер телефона (например, 7075602) или электронный адрес – *E-Mail* (напри-

мер, *FedorenkoYP@Mail.ru*), если же телефон или *E-Mail* у сотрудника отсутствует, то в этом столбце ставится буква *н* (от слова *нет*). Число строк в файле совпадает с числом сотрудников учреждения, а число столбцов – с числом всех полей используемой переменной структурного типа. Ниже показан пример указанной базы данных, состоящий всего лишь из трёх строк. Из-за полиграфических ограничений каждая строка файла представлена на *двух* строках страницы.

Содержимое файла *Baza.txt*

```
Федоренко  Юрий      Петрович      62461 Пивденное  Артема        45/2        1
снс      813,0  н

Шевченко   Тарас      Григорьевич  61077 Харьков   Пушкина       112         2
инж      780 Taras.G.Shevchenko@Univer.Kharkov.ua

Коба       Дарья      Михайловна   66666 Ракитное    Хмельницкого  99/2        66
зав      966      1369696
```

При обработке баз данных первостепенное значение имеет количество позиций для размещения отдельного поля (или длина поля) и номер начала позиции каждого такого поля.

В нашей базе указанные параметры полей приведены в следующей таблице.

Имя поля	Позиция начала	Длина поля
Фамилия	1	11
Имя	12	9
Отчество	21	12
Индекс	33	6
Город	39	11
Улица	50	13
Номер дома	63	7
Номер квартиры	70	4
Должность	74	5
Оклад	79	8
Телефон или <i>E-Mail</i>	87	50

Интерфейс приложения показан на рис. 13.1. Программа предложит пользователю ввести фамилию сотрудника, индивидуальные данные которого требуется узнать. Перечень возможных для ознакомления индивидуальных характеристик сотрудника приведен в списке выбора (его поле находится выше командных кнопок интерфейса). При вводе фамилии, которая не значится в базе, приложение выведет сообщение: *Отсутствует в списке сотрудников*. В этом случае кнопка интерфейса *Показать* окажется неактивной, поэтому выбрать какую-либо характеристику из списка окажется *невозможным*. Если же, не выбрав строку (из

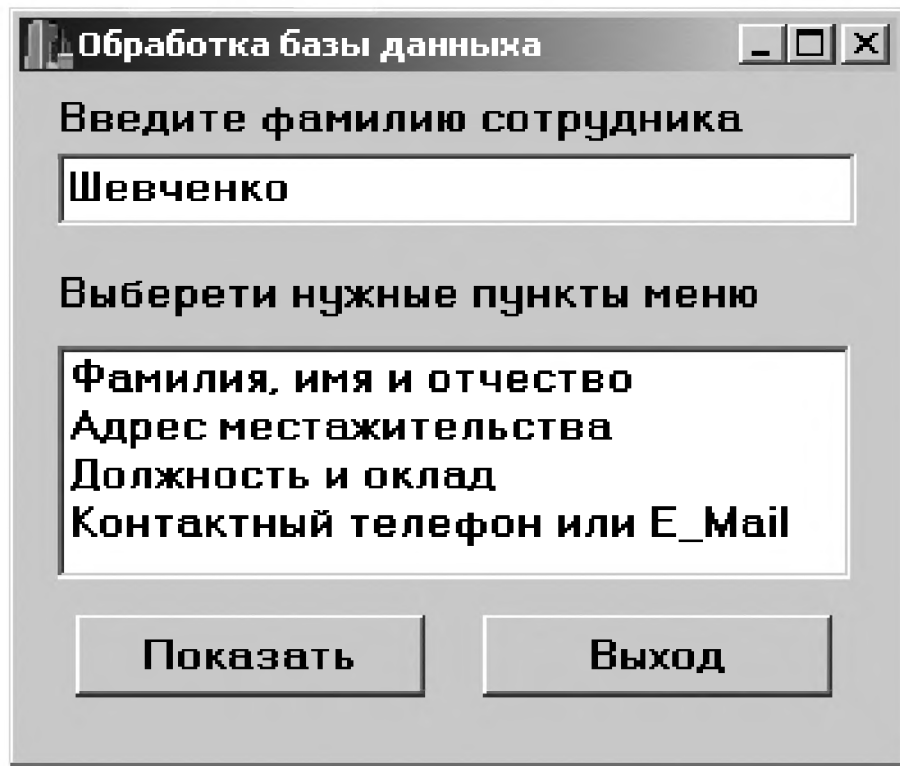


Рис. 13.1. Интерфейс приложения *Обработка базы данных*

списка выбора), нажать кнопку *Показать*, то появится предупреждение: *Ни один пункт не выбран*. Только, когда требуемая фамилия *имеется* в базе данных, после выбора (при помощи указателя мыши) желаемой строки из предлагаемого списка характеристик и нажатия кнопки *Показать*, выводится затребованная информация.

Программы по обработке таких и аналогичных баз данных могут решать большой круг задач, например: выводить из общего списка список сотрудников заданных профессий или категорий (научных сотрудников, инженеров, техников, лаборантов и др.); список сотрудников в указанном интервале возрастов; номера машин сотрудников, если они у них имеются, и многое, многое другое.

По понятным причинам задача учебной программы максимально упрощена. Основные её элементы рассмотрены ранее. В функции *FormCreate* осуществляется запись базы данных из файла в переменную *Spis_Isx* и подсчитывается *фактическое* число сотрудников *iCh_Sotr* в базе данных. Эта работа осуществляется вызовом пользовательской функции *Vvod*. Затем, при помощи пользовательской функции *Zapoln_Strt*, база данных копируется из *Spis_Isx* в массив структур *Sotrud* типа *strt_Sotr*. В результате *все* поля структур каждого сотрудника оказываются заполненными.

В структуру *strt_Sotr* включено вариантное поле с именем *Tel_EMail*. Это поле имеет тип *unsigned long*, если сотрудник указал для связи с ними контактный телефон, и массив *char* в случае, когда связь осуществляется при помощи электронной почты *E-Mail*. Чтобы программа могла определить тип данных этого поля для *конкретного* сотрудника (это важно как для чтения, так и записи информации), в структуре *strt_Sotr* предусмотрено специальное поле *перечислимого* типа с име-

нем *Kom* (коммутатор). Задание *фактического* числа литер электронного адреса в структуре *strt_Sotr* осуществляется полем *iDli* с целочисленным типом *int*.

По завершению ввода фамилии в поле ввода *Vvod_Fam* функция *Vvod_FamExit* (событие *OnExit*) при помощи вызова пользовательской функции *Poisk* осуществляет *поиск* требуемой фамилии: при успешном поиске она возвращает номер сотрудника *iNom_Sotr* с искомой фамилией в базе данных, а при отрицательном – делает недоступной кнопку *Показать* и выводит сообщение *Отсутствует в списке сотрудников*.

Пользователь при наборе фамилии может перед фамилией и после неё набрать несколько пробелов, однако это не повлияет на результат поиска, поскольку после окончания набора фамилии *все* пробелы впереди и позади введенного слова устраняются стандартной функцией *Trim*.

Нажатие кнопки *Показать* (с именем *Pokaz*) обрабатывает функция *PokazClick*. Именно эта функция, вызывая методы структуры, производит показ затребованных характеристик сотрудника. Если ни один из пунктов не выбран, то нажатие кнопки *Показать* выводит сообщение: *Ни один пункт не выбран*.

Вначале лишь просмотрите код и комментарии приложения. Затем *настоятельно* рекомендуем *детально* изучить текст программы. Это поможет освоить применение структур и легко решить задачи, предложенные в конце урока для самостоятельного решения.

Код приложения *Обработка базы данных*

```
const int iChSotr= 30;// Ориентировочное число сотрудников
int iCh_Sotr;//Фактическое число сотрудников в базе данных
int iNom_Sotr;//Номер искомого сотрудника в списке
TStringList *Spis_Isx= new TStringList;
String Isx_f= "Baza.txt";//Файловая переменная
//Используется для заполнения вариантного поля структуры
enum Kont{tel, e_mail, net};

void Vvod(int & iCh_Sotrydn)
{
    try
    {
        Spis_Isx->LoadFromFile(Isx_f);
    }//try
    catch (...)
    {
        ShowMessage("Файл \" " + Isx_f + " \" не найден");
        Abort();
    }//catch
    iCh_Sotrydn= Spis_Isx->Count;
} //Vvod

//Используется в вариантном поле структуры
union Var_Un
{
    unsigned long Telef;//Для хранения номера телефона
    char EMail[50];//Адрес электронной почты
}; //Var_Un
```

```

struct strt_Adres
{
    unsigned long Ind;
    String Gorod, Yl, Nom_Dom;
    short Nom_Kv;
}; //strt_Adres

struct strt_Rabota
{
    String Dolgn;
    float Oklad;
}; //strt_Rabota

struct strt_Sotr
{
    String Fam, Im, Otch;
    strt_Adres Adres ;
    strt_Rabota Rabota;
    Kont Kom; //tel, e_mail, net
    Var_Un Tel_EMail;
    int iDli; //Число литер в E-Mail

    void Pokaz_FIO(void); //Прототипы методов
    void Pokaz_Adres(void);
    void Pokaz_Dolgn_Oklad(void);
    void Pokaz_Tel_EMail(void);
} Sotrudn[iChSotr]; //Массив структур

void strt_Sotr::Pokaz_FIO(void)
{
    ShowMessage(Fam + " " + Im + " " + Otch);
} //Pokaz_FIO//;

void strt_Sotr::Pokaz_Adres(void)
{
    ShowMessage(IntToStr(Adres.Ind) + ", г. " + Adres.Gorod +
        ", ул. " + Adres.Yl + ", д. " + Adres.Nom_Dom + ", кв. " +
        IntToStr(Adres.Nom_Kv));
} //Pokaz_Adres;

void strt_Sotr::Pokaz_Dolgn_Oklad(void)
{
    ShowMessage(Rabota.Dolgn + ", оклад: "
        + FloatToStrF(Rabota.Oklad, ffFixed, 8, 2) + " гривень");
} //Pokaz_Dolgn_Oklad;

void StrToMasChar(String Stroka, char MassChar[50], int iDl)
{
    //Преобразует переменную типа String в массив типа char
    for (int ii= 1; ii<= iDl; ii++)
        MassChar[ii - 1]= Stroka[ii];
} //StrToMasChar

String MasCharToStr(char MassChar[], int iDl)
{
    //Преобразует массив типа char в переменную типа String
    String Stroka= "";
    for (int ii= 1; ii<= iDl; ii++)
        Stroka= Stroka + MassChar[ii - 1];
    return Stroka;
}

```

```
//MasCharToStr

void strt_Sotr::Pokaz_Tel_EMail(void)
{
    switch (Kom)
    {
        case net: ShowMessage("У " + Fam + " имеется только почтовая связь!");
            break;
        case tel: ShowMessage(Tel_EMail.Telef);
            break;
        case e_mail:
            String Stro= MasCharToStr(Tel_EMail.EMail, iDli);
            ShowMessage(Stro);
    }
}

//Pokaz_Tel_Email

void Zapoln_Strt(int iChi_Sotr)
{
    String Stroka;
    for (int ij= 0; ij< iChi_Sotr; ij++)
    {
        String Stroka= Spis_Isx->Strings[ij];
        Sotrudn[ij].Fam= Trim(Stroka.SubString(1, 11));
        Sotrudn[ij].Im= Trim(Stroka.SubString(12, 9));
        Sotrudn[ij].Otch= Trim(Stroka.SubString(21, 12));
        String Stroka_1= Stroka.SubString(33, 6);
        Sotrudn[ij].Adres.Ind= StrToInt(Stroka_1.Trim());
        Sotrudn[ij].Adres.Gorod= Trim(Stroka.SubString(39, 11));
        Sotrudn[ij].Adres.Yl= Trim(Stroka.SubString(50, 13));
        Sotrudn[ij].Adres.Nom_Dom= Trim(Stroka.SubString(63, 7));
        Stroka_1= Stroka.SubString(70, 4);
        Sotrudn[ij].Adres.Nom_Kv= StrToInt(Stroka_1.Trim());
        Sotrudn[ij].Rabota.Dolgn=Trim(Stroka.SubString(74, 5));
        Stroka_1= Stroka.SubString(79, 8);
        Sotrudn[ij].Rabota.Oklad= StrToFloat(Stroka_1.Trim());
        Stroka_1= Trim(Stroka.SubString(87, 50));
        int iS= 0; //Счётчик числа нецифровых литер в Stroka_1
        //Цикл по литерам Stroka_1
        for (int ik= 1; ik<= Stroka_1.Length(); ik++)
            if (Stroka_1[ik]< '0' || Stroka_1[ik]> '9')
                iS++;
        if (iS== 0) //Все символы являются цифровыми
        {
            Sotrudn[ij].Kom= tel;
            Sotrudn[ij].Tel_EMail.Telef= StrToInt(Stroka_1);
            Sotrudn[ij].iDli= 0;
        } //if_1
        else
            if (Stroka_1!= "н") //или (Stroka_1.Length()>1)
            {
                Sotrudn[ij].Kom= e_mail;
                Sotrudn[ij].iDli= Stroka_1.Length();
                StrToMasChar(Stroka_1,
                    Sotrudn[ij].Tel_EMail.EMail, Sotrudn[ij].iDli);
            } //if_2
        else
    }
```

```

        Sotrudn[ij].Kom= net;
    } //for_int
} //Zapoln_Strt

void Poisk(int &iNom_Sotrydn, String Fami)
{
    //Поиск введенной фамилии в базе данных
    Form1->Pokaz->Enabled= false;
    for (int ij= 0; ij< Spis_Isx->Count; ij++)
        if (Fami==Trim(Sotrudn[ij].Fam))
        {
            iNom_Sotrydn= ij;
            Form1->Pokaz->Enabled= true;
            return; //Если поиск завершился успешно
        } //if
    ShowMessage("Отсутствует в списке сотрудников");
} //Poisk

void __fastcall TForm1::PokazClick(TObject *Sender)
{
    //Номер выделенной строки меню
    int iSelekt= ListBox1->ItemIndex;
    switch (iSelekt)
    {
        case 0: Sotrudn[iNom_Sotr].Pokaz_FIO();
            break;
        case 1: Sotrudn[iNom_Sotr].Pokaz_Adres();
            break;
        case 2: Sotrudn[iNom_Sotr].Pokaz_Dolgn_Oklad();
            break;
        case 3: Sotrudn[iNom_Sotr].Pokaz_Tel_EMail();
            break;
        default: //или case -1:
            ShowMessage("Ни один пункт не выбран");
    } //switch
} //PokazClick

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    //Данные базы копируются из файла в переменную Spis_Isx
    //Подсчёт фактического числа строк базы данных
    Vvod(iCh_Sotr);
    //Заполняем поля структуры при помощи переменной Spis_Isx
    Zapoln_Strt(iCh_Sotr);
} //FormCreate

void __fastcall TForm1::VuxodClick(TObject *Sender)
{
    delete Spis_Isx;
    Close();
} //VuxodClick

void __fastcall TForm1::Vvod_FamExit(TObject *Sender)
{
    //Ввод фамилии сотрудника
    String Famil= Vvod_Fam->Text;
    Famil= Trim(Famil);
}

```



```
//Определяем номер сотрудника в списке или устанавливаем  
//факт его отсутствия в базе данных  
Poisk(iNom_Sotr, Famil);  
} //Vvod_FamExit
```

В этом приложении знакомимся с ещё одним очень важным визуальным объектом типа *TListBox* – списком строк (закладка *Палитры Компонентов Standard*). Поле для выбора его строк показано на рис. 13.1 над командными кнопками, имя использованного экземпляра – *ListBox1*. У объекта такого типа имеется много свойств, методов и событий. Однако рассмотрим лишь некоторые из них. В списке выбора обычно *только выбирают* строку или строки, ввод информации в строки в процессе работы приложения возможен, однако применяется редко.

Если требуется реализовать выбор только *одной* строки, то свойству *MultiSelect* (выбор нескольких строк) следует оставить значение *false*, установленное по умолчанию. Для осуществления выбора одной и более строк – этому свойству требуется назначить значение *true* (в *Инспекторе Объектов* либо программно). В учебной задаче рассмотрена возможность выбора только одной строки списка. Такой выбор можно осуществить с применением клавиши *Tab* (осуществляющей перемещение фокуса активности по всем объектам управления интерфейса) и клавиш перемещения курсора *Вверх* и *Вниз*. Однако более удобным, на наш взгляд, является использование указателя мыши.

Строки списка нумеруются, начиная с нуля. Номер выбранной строки в ходе выполнения программы записывается в свойстве *ItemIndex*. Если ни одна строка не выбрана, то в этом свойстве хранится –1. В нашем приложении обработка выбранной строки списка осуществляется функцией *PokazClick*: в переменную-селектор *iSelekt* записывается номер выделенной строки, после чего оператор *switch* (*iSelekt*) вызывает соответствующий метод структуры *strt_Sotr*, предназначенный для обработки заданного варианта, либо выводит сообщение: *Ни один пункт не выбран*.

Для осуществления возможности выбора *нескольких* строк (или сразу всех) прежде всего тело функции *FormCreate* следует дополнить строкой:

```
Form1->ListBox1->MultiSelect= true;
```

Эту же строку можно расположить и в конструкторе, соответствующая операция осуществима также в *Инспекторе Объектов*. После использования любого из приведенных вариантов проверяется, выбрана или нет заданная строка списка. Для этого используется свойство *Selected[Nom_Vubran_Stroki]*. В свойстве *ItemIndex* в этом случае хранится номер строки, находящейся в фокусе ввода (эта строка обведена рамкой из жёлтых точек). В случае реализации выбора *нескольких* строк функция *PokazClick* может иметь, например, такой вид:

```
void __fastcall TForm1::PokazClick(TObject *Sender)  
{  
    for (int ij= 0; ij< ListBox1->Items->Count; ij++)  
        if (ListBox1->Selected[ij])  
            switch (ij)  
            {
```

```

    case 0:Sotr[iNom_Sotr].Pokaz_FIO();
        break;
    case 1:Sotr[iNom_Sotr].Pokaz_Adres();
        break;
    case 2:Sotr[iNom_Sotr].Pokaz_Dolgn_Oklad();
        break;
    case 3:Sotr[iNom_Sotr].Pokaz_Tel_EMail();
        break;
    default:
        ShowMessage("Ни один пункт не выбран");
} //switch
} //PokazClick

```

В функции *PokazClick* использовано подсвойство *Count* свойства *Items* (список строк). В этом подсвойстве хранится фактическое число строк в списке. Свойство *Items* имеет уже использованный ранее тип *TStrings*. Строки этого списка (см. рис.13.1) записываются в процессе разработки программы при помощи *Инспектора Объектов*: выбрать свойство *Items*, нажать кнопку с тремя точками для входа в редактор этого свойства, в окне ввода набрать требуемые строки, после чего нажать кнопку *ОК*. Очистить список или добавить в него новую строку можно также программно, пользуясь методами класса *TStrings*, соответственно *Clear* и *Add*:

```

ListBox1->Items->Clear();
ListBox1->Items->Add("Добавлена строка в конец списка");

```

Метод *Add* добавляет строку в конец списка и возвращает индекс добавленной строки. Для свойства *Items* применимы также и другие методы и свойства класса *TStrings*.

13.4.7. Наследование структур

В ходе тестирования приложений выявляются их отдельные недостатки, появляется ряд предложений по дополнению списка решаемых задач. Для быстрой модернизации программ, направленной на расширение её функциональных возможностей можно не изменять те её части, которые надёжно и эффективно работают в программе-прототипе. Это оказывается возможным при помощи наследования структур и классов, позволяющего максимально *сократить* срок модернизации приложений.

Проиллюстрируем механизм наследования структур на прежней учебной задаче. Предположим, вначале разработана программа, в которой имеются две структуры *strt_Adr* и *strt_Rab* со *своими* полями и методами. Разработаем структуру *strt_Sotr*, которая наследует (включает в себя) все поля и методы упомянутых структур, и имеет *дополнительные* поля и методы, позволяющие программе выводить и другие характеристики сотрудника. Эта структура называется структурой-наследницей, *дочерней* структурой или структурой-потомком. Те же структуры, которые наследует дочерняя структура, именуется *родительскими* или предками. Для создания дочерней структуры после её имени *strt_Sotr* за двоеточием необходимо перечислить через запятую родительские структуры

strt_Adr и *strt_Rab*. При обращении к полям и методам дочерней структуры, впитавшей в себя поля и методы наследованных структур, родительские структуры *не упоминаются* (что сокращает длину кода при обращении к выбранному полю). Ниже приведены только те структуры и функции предшествующего приложения, которые претерпели изменения в связи с иллюстрацией механизма наследования структур (помимо сокращения длины отдельных идентификаторов).

```
struct strt_Adr
{
    unsigned long Ind;
    String Gorod, Yl, Nom_Dom;
    short Nom_Kv;
    void Pokaz_Adr(void);
}; // strt_Adr

struct strt_Rab
{
    String Dolg;
    float Okl;
    void Pokaz_Dolg_Okl(void);
}; // strt_Rab

struct strt_Sotr:strt_Adr, strt_Rab
{
    String Fam, Im, Otch;
    Kont Kom; // tel, e_mail, net
    Var_Un Tel_EMail;
    int iDli; // Число литер в E-Mail
    void Pokaz_FIO(void); // Прототипы
    void Pokaz_Tel_EMail(void);
} Sotrud[iChSotr]; // Массив структур

void strt_Sotr::Pokaz_FIO(void)
{
    ShowMessage(Fam + " " + Im + " " + Otch);
} // Pokaz_FIO//

void strt_Adr::Pokaz_Adr(void)
{
    ShowMessage(IntToStr(Ind) + ", г. " + Gorod + ", ул. " +
        Yl + ", д. " + Nom_Dom + ", кв. " + IntToStr(Nom_Kv));
} // Pokaz_Adres;

void strt_Rab::Pokaz_Dolg_Okl(void)
{
    ShowMessage(Dolg + ", оклад: "
        + FloatToStrF(Okl, ffFixed, 8, 2) + " гривень");
} // Pokaz_Dolg_Okl;

void Zapoln_Strt(int iChi_Sotr)
{
    String Stroka;
    for (int ij= 0; ij< iChi_Sotr; ij++)
    {
        String Stroka= Spis_Isx->Strings[ij];
        Sotrud[ij].Fam= Trim(Stroka.SubString(1, 11));
        Sotrud[ij].Im= Trim(Stroka.SubString(12, 9));
```

```

Sotrud[ij].Otch= Trim(Stroka.SubString(21, 12));
String Stroka_1= Stroka.SubString(33, 6);
Sotrud[ij].Ind= StrToInt(Stroka_1.Trim());
Sotrud[ij].Gorod= Trim(Stroka.SubString(39, 11));
Sotrud[ij].Yl= Trim(Stroka.SubString(50, 13));
Sotrud[ij].Nom_Dom= Trim(Stroka.SubString(63, 7));
Stroka_1= Stroka.SubString(70, 4);
Sotrud[ij].Nom_Kv= StrToInt(Stroka_1.Trim());
Sotrud[ij].Dolg=Trim(Stroka.SubString(74, 5));
Stroka_1= Stroka.SubString(79, 8);
Sotrud[ij].Ok1= StrToFloat(Stroka_1.Trim());
Stroka_1= Trim(Stroka.SubString(87, 50));
int iS= 0;//Счётчик числа нецифровых литер в Stroka_1
//Цикл по литерам Stroka_1
for (int ik= 1; ik<= Stroka_1.Length(); ik++)
    if (Stroka_1[ik]< '0' || Stroka_1[ik]> '9')
        iS++;
if (iS== 0)//Все символы являются цифровыми
{
    Sotrud[ij].Kom= tel;
    Sotrud[ij].Tel_EMail.Telef= StrToInt(Stroka_1);
    Sotrud[ij].iDli= 0;
}//if_1
else
    if (Stroka_1!= "н")//или (Stroka_1.Length()>1)
    {
        Sotrud[ij].Kom= e_mail;
        Sotrud[ij].iDli= Stroka_1.Length();
        StrToMasChar(Stroka_1,
        Sotrud[ij].Tel_EMail.EMail, Sotrud[ij].iDli);
    }//if_2
    else
        Sotrud[ij].Kom= net;
}//for_ij
}//Zapoln_Strt

```

13.4.8. Защита полей и методов структур

При коллективной работе над объёмными приложениями (более 10 – 100 тыс. строк), большом числе введенных переменных и функций со сходными названиями, предназначенными для решения близких задач, вероятность ошибок возрастает. Особенно опасна ситуация, когда применённая ошибочно функция выполняет похожие задачи и возвращает правдоподобный результат. Для борьбы с такими ошибками отдельные поля и методы структур делают доступными *только* внутри данной структуры, или доступными в данной структуре и *во всех* её *потомках*. Для знакомства с этими механизмами защиты вновь привлечём прежнюю задачу.

Модернизируем структуры *strt_Adr*, *strt_Rab* и *strt_Sotr*.

```

struct stre_Adr
{
    public: //Общедоступные поля и методы

```

```

    unsigned long Ind;
    String Gorod, Yl, Nom_Dom;
    short Nom_Kv;
    protected://Доступно в данной структуре и у её потомков
        void Pokaz_Adres(void);
}; //strt_Adr

struct strt_Rab
{
    String Dolg;
    float Okl;
    protected://Доступно в данной структуре и у её потомков
        void Pokaz_Dolg_Okl(void);
}; //strt_Rabota

struct strt_Sotr:strt_Adr, strt_Rab
{
    String Fam, Im, Otch;
    Kont Kom;//tel, net , e_mail
    Var_Un Tel_EMail;
    private://Доступно только в данной структуре
        void Pokaz_FIO(void); //Прототипы
        void Pokaz_Tel_EMail(void);
    public:
        void Pokaz(int iSelektor);
} Sotrud[iChSotr]; //Массив структур

```

Служебное слово *public* (общедоступный) в структуре *strt_Adr* делает *открытыми* (доступными для внешнего обращения) все поля и методы, расположенные ниже него вплоть до другого ключевого слова *protected* (в нашем примере методы отсутствуют). По умолчанию (отсутствует какое либо ключевое слово) все элементы структуры являются *открытыми*. Именно по этой причине в предшествующем примере функции *Zapoln_Strykt* позволено заполнить данными *все* поля структуры (взятыми из переменной *Spisok_Isx*), а функции *PokazClick* – вызывать различные методы структурной переменной *Sotrud*. В структуре *strt_Rab* нет никакого ключевого слова впереди её полей, поэтому её поля *Dolg* и *Okl* общедоступны.

Применение ключевого слова *protected* (защищённый) в структурах *strt_Adr* и *strt_Rab* приводит к тому, что методы (и поля, если бы они в примере были), расположенные ниже этого слова (за двоеточием), оказываются *недоступными* вне структуры. Их можно применять только в *указанных* структурах (в переменных этих структурных типов) и в их наследниках (в дочерних структурах, в нашем примере в структуре *strt_Sotr*).

Слово *private* (закрытый), использованное в структуре *strt_Sotr*, позволяет методы *Pokaz_FIO* и *Pokaz_Tel_EMail* применять *только* в этой структуре. Если бы структура *struct_Sotr* наследовалась какой-то иной структурой, то упомянутые методы в ней были бы *недоступны*. В структуре *strt_Sotr* вновь применяется служебное слово *public*. Это сделано для того, чтобы указать *нижнюю* границу области действия ключевого слова *private*. Поэтому в структуре *strt_Sotr* метод *Pokaz* является общедоступным. Поскольку в этой структуре все другие методы

являются защищёнными, то их вызов можно осуществить только общедоступным методом *Pokaz*, который *специально* разработан для этой цели. Прежний вариант функции *PokazClick* в данном случае будет непригодным, его необходимо изменить для осуществления возможности вызова защищённых методов. Реализация метода *Pokaz* и нового варианта функции *PokaClick* приведены ниже.

```
void strt_Sotr::Pokaz(int iSelektor)
{
    switch (iSelektor)
    {
        case 0:Sotr[iNom_Sotr].Pokaz_FIO();
            break;
        case 1:Sotr[iNom_Sotr].Pokaz_Adr();
            break;
        case 2:Sotr[iNom_Sotr].Pokaz_Dolg_Okl();
            break;
        case 3:Sotr[iNom_Sotr].Pokaz_Tel_EMail();
            break;
        default:
            ShowMessage("Ни один пункт не выбран");
    } //switch
} //Pokaz

void __fastcall TForm1::PokazClick(TObject *Sender)
{
    for (int ij= 0; ij< ListBox1->Items->Count; ij++)
        if (ListBox1->Selected[ij])
            Sotr[iNom_Sotr].Pokaz(ij);
} //PokazClick
```

Если таким *сложным* образом вызывать методы, предназначенные для обработки конкретных полей структур, то, скорее всего, их применение глубоко осознано, действительно необходимо по ходу выполнения программы. Поэтому такой способ обработки полей структур повышает надёжность приложений.

13.4.9. Двухнаправленный список структур

Двухнаправленный или *двухсвязный* список структур относится к динамическому способу размещения данных. При использовании массива структур объём оперативной памяти, необходимый для размещения *всех* его элементов, выделяется на этапе *компиляции* программы. Динамические же структуры данных могут *изменять* число своих элементов *по ходу работы* программы. Поэтому оперативная память при использовании динамических структур расходуется наиболее оптимально: выделяется ровно столько элементов списка, сколько требуется в данный момент, а не с запасом, как в случае применения массива структур.

Списки структур могут быть также *однонаправленными (односвязными)*, частными их случаями являются *стеки, кольцевые списки и очереди*. Важными для задач поиска являются динамические структуры данных, которые называются *бинарными деревьями*. Рассмотрение перечисленных видов динамических структур выходит за рамки настоящего пособия.

Все динамические структуры данных основываются на применении указателей. Поэтому структуры – элементы этих образований, могут занимать *несмежные* блоки оперативной памяти. Важность способов хранения информации, реализованных в динамических структурах, трудно переоценить. На их основе разработан целый ряд очень эффективных алгоритмов, которые находят применение практически во всех стандартных приложениях ведущих фирм.

Следует отметить, что *C++Builder* позволяет *большинство* задач, связанных с динамическими структурами, решать при помощи типов данных *TList* и *TStringList*. В этих классах (эти типы данных реализованы, как классы) имеется много очень удобных методов, позволяющих выполнять практически все операции по управлению списками динамических структур, не задумываясь о механизмах, положенных в их основу. На предшествующих уроках упомянутые классы (и их методы) уже использовались и читатели, видимо, успели ощутить их большие возможности. Однако программисты, чтобы не остаться *ремесленниками*, просто *обязаны* владеть этими скрытыми механизмами и уметь применять их не только посредством использования упомянутых классов *Builder*. По этой причине *настоятельно* рекомендуем детально разобраться в материале настоящего пункта, поскольку в нём разъясняются алгоритмы *очень полезные* при разработке многих программ. Не огорчайтесь, если при *первом* чтении что-то окажется не совсем ясным. Повторное чтение *обязательно* прояснит ситуацию. Излагаемые алгоритмы не настолько просты, чтобы открыть свою сущность при первом (особенно беглом) чтении.

Как отмечалось, динамические структуры данных непросты в понимании, поэтому при их изучении прибегнем к графическим иллюстрациям основных фрагментов кода. Если читатель не очень внимательно изучил указатели, ссылки и способы передачи параметров в пользовательские функции, то настоятельно рекомендуем вначале разобраться в перечисленных вопросах: без их глубокого уяснения дальнейшее чтение не имеет смысла. И, наконец, последняя рекомендация-требование: *все* фрагменты кодов *необходимо* реализовывать на компьютере. В принципе, эта рекомендация актуальна при изучении *всех* разделов, однако при рассмотрении списков её никак нельзя игнорировать, без её выполнения полное понимание материала окажется, скорее всего, невозможным и предлагаемые алгоритмы *никогда* не появятся в ваших программах.

Создадим двунаправленный список данных о сотрудниках учреждения. Пусть данные отдельного сотрудника описываются структурой *strt_Sotr*, использованной в п. 13.4.7. Предположим (для сокращения процедуры ввода), что данные всех сотрудников *уже* занесены в массив структур *Sotrud* типа *strt_Sotr*, также применённый в упомянутом пункте. Для связи одной структуры разрабатываемого двунаправленного списка с другой введём структуру следующего типа:

```
struct Spis_Sotr
{
    Spis_Sotr *Levuj;
    strt_Sotr Sotr; // Данные о сотруднике, далее просто данные
    Spis_Sotr *Pravuj;
```

```
}; // Spis_Sotr
```

В этой структуре имеется всего *три* поля. Поле с именем *Sotr* имеет тип *strt_Sotr*, в нём описываются все индивидуальные характеристики отдельного сотрудника. Ранее отмечалось, что тип поля структуры *не может* совпадать с типом самой структуры, однако разрешается, чтобы поле структуры было указателем на структурный тип, в котором это поле описано. Поэтому объявление полей *Levuј* и *Pravuј* типа *Spis_Sotr** вполне правомерно. Напоминаем, что в переменную-указатель (в нашем случае поле-указатель) можно записать *только* адрес переменной заданного типа. Поэтому в полях *Levuј* и *Pravuј* разрешается хранить лишь адреса переменных-структур типа *Spis_Sotr*. При формировании списка (или связанного набора) структур в *каждой* структуре теперь имеется возможность хранить адреса своих *смежных* соседей: в поле *Levuј* будем хранить адрес структуры, расположенной слева, а в поле *Pravuј* – справа от заданной структуры.

Для *формирования* списка структур типа *Spis_Sotr* объявим три указателя:

```
Spis_Sotr *Nachalo= NULL, //Указатель на начало списка
          *Konec= NULL, //Указатель на конец списка
          *Novuј; //Указатель на очередную структуру списка
```

Несколько позже эти указатели направим соответственно в начало, в конец списка и на очередную новую структуру, вставляемую в список структур. Поскольку списка *ещё нет* его указатели начала и конца направлены в *NULL* (на несуществующий адрес).

Создадим теперь *первый* элемент списка при помощи функции *Pervuј*. Эта функция порождает структуру типа *Spis_Sotr* и возвращает указатель на неё. В структуре, порождённой этой функцией, информационное поле замещается единственным аргументом функции *Sotrudnichek*:

```
//Формирование первой структуры
Spis_Sotr * Pervuј(strt_Sotr Sotrudnichek)
{
    Novuј = new Spi_Sotr;
    Novuј->Levuј= NULL;
    Novuј->Sotr= Sotrudnichek;
    Novuј->Pravuј= NULL;
    return Novuј;
} //Pervuј
```

Далее используем функцию *Pervuј* в функции *Spisok_1*, формирующей список из *единственной* структуры:

```
//Формирование списка из единственной структуры
void Spisok_1(void)
{ //Начальная структура массива
    Nachalo= Pervuј(Sotrud[0]); //Адрес первой структуры
    //Список начинается и заканчивается первым элементом
    Konec= Nachalo;
} //Spisok_1
```

В функции *Spisok_1* для заполнения информационного поля структуры типа *Spis_Sotr* использована начальная структура упомянутого выше массива структур *Sotrud*. Выполнение первой строки этой функции иллюстрирует рис. 13.2.

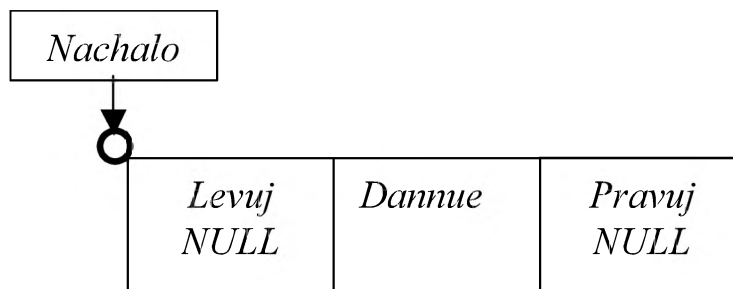


Рис. 13.2. Формирование первого элемента списка

Здесь *Nachalo* – имя указателя с адресом структуры, на которую он направлен, *Levuj* и *Pravuj* – соответствующие поля-указатели упомянутой структуры, *Dannue* – это информационное поле структуры *Nachalo* с индивидуальными данными сотрудника (в нашем случае это данные о сотруднике, приведенные в начале массива *Sotrud*). Поскольку элемент списка является единственным, то поля указатели этого элемента направлены в *NULL*: они ещё не имеют соседей (смежных элементов) *ни справа, ни слева*. Полный кружок в левом верхнем углу структуры представляет собой адрес начальной ячейки объекта структуры, то есть адрес той ячейки в памяти компьютера, начиная с которой единым блоком расположены все поля структурной переменной. Поэтому на эту ячейку направлен указатель (стрелка) с именем *Nachalo*.

Вторая строка функции завершает формирование списка из единственной структурной переменной – единственного элемента списка. Список из единственной структуры в графическом изображении показан на рис. 13.3.

Как видите, в списке из одной структуры на её начало направлены оба указателя списка: *Nachalo* и *Konec*.

При помощи функции *Vklyuchenie* в конец уже *существующего* списка можно добавлять новые структуры.

```
//Добавление новой структуры в конец списка структур
void Vklyuchenie(Spis_Sotr **Konec_Spiska, strt_Sotr Sotrudnichek)
{
    Novuj = new Spis_Sotr;
    Novuj->Levuj= *Konec_Spiska;
    Novuj->Sotr= Sotrudnichek;
    Novuj->Pravuj= NULL;
    (*Konec_Spiska)->Pravuj= Novuj;
    *Konec_Spiska= Novuj;
} //Vklyuchenie
```

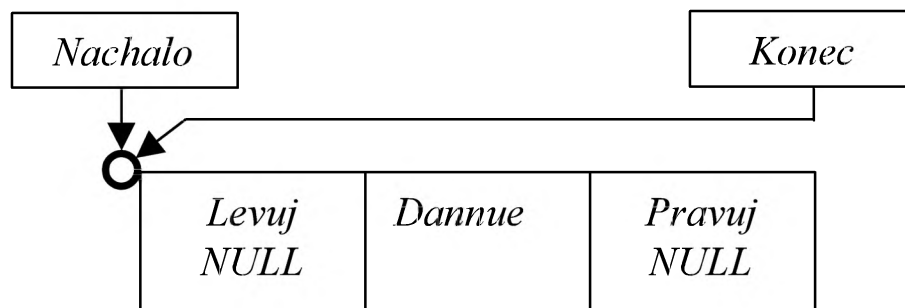


Рис. 13.3. Список, состоящий из единственной структуры

У этой функции имеется *два* параметра. Второй параметр предназначен для полнения информационного поля вновь созданной структуры $Novuj \rightarrow Sotr = trudnichek$. Первым параметром функции *Vklyuchenie* является указатель на указатель с формальным именем *Konec_Spiska*. При вызове функции этот параметр следует заместить *адресом* указателя *Konec*, а не адресом, хранимым в его объекте (в нашем случае – адресом указателя последней структуры). Этот параметр передаётся в функцию *Vklyuchenie по адресу*, следовательно, его значение (объектная часть указателя-параметра), изменяемое в функции, *автоматически* терпит изменение и в фактическом параметре.

Другими словами можно сказать ещё и так: адрес ячейки, в которой записан адрес конечной структуры, *неизменен*, а адрес конечного элемента изменяется в ходе добавления новых структур в формируемый список. Поэтому в ячейку *Konec* последовательно будут записываться адреса от первой структуры до последней структуры списка. Давайте рассмотрим, как же этот процесс выполняет при помощи функции *Vklyuchenie*. После создания *новой* структуры с именем *Novuj*, в её поле *Levuj* записывается адрес *объекта* указателя *Konec_Spiska*: $Novuj \rightarrow Levuj = *Konec_Spiska$. Это сделано потому, что новая структура с именем *Novuj* вставляется *после* последней структуры списка, адрес которой хранится в указателе *Konec_Spiska*. Теперь этот адрес записан в поле *Levuj* вставляемой структуры *Novuj*.

Поскольку новая структура вставляется в *конец* существующего списка структур, её полю-указателю с именем *Pravuj* присваивается *NULL*, что является знаком *последнего* элемента списка: $Novuj \rightarrow Pravuj = NULL$. Далее в функции *Vklyuchenie* в поле-указатель *Pravuj* последней структуры *предшествующего* варианта списка записывается адрес вновь созданной структуры *Novuj*: $Konec_Spiska \rightarrow Pravuj = Novuj$. Этот этап дополнения списка структур новым (горячим) элементом иллюстрируется на рис. 13.4, где формальный параметр рассмотренной функции *Konec_Spiska* заменен фактическим параметром *Konec*, использованным при её вызове (см. ниже).

Последняя строка функции *Vklyuchenie* завершает создание списка из двух структур: в объектную часть указателя *Konec_Spiska* записывается адрес *новой* структуры *Novuj*: $*Konec_Spiska = Novuj$. Графический вариант списка из двух структур в окончательном виде показан на рис. 13.5.

Для создания списка из начального (нулевого) элемента массива *Sotrud* и *последовательного* включения в этот список всех оставшихся элементов массива

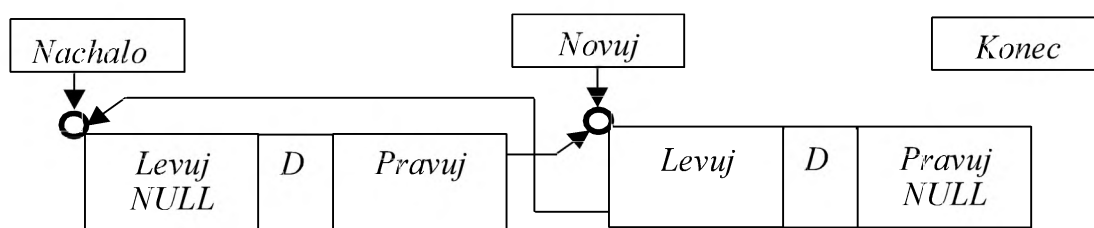


Рис. 13.4. Этап создания списка из двух структур

Levuj
NULL

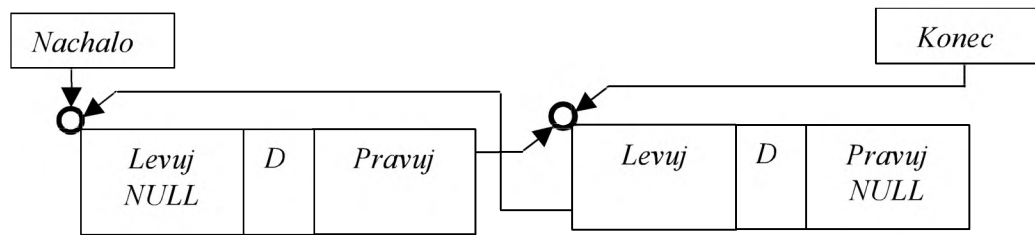


Рис. 13.5. Список из двух структур

Sotrud (начиная с первого) предназначена функция *SozdatjSpisokClick*. В этой функции элементы массива *Sotrud* от *первого* (но не нулевого) до *последнего* включаются в список при помощи вызова функции *Vklyuchenie* в теле оператора *for*:

```
void __fastcall TForm1::SozdatjSpisokClick(TObject *Sender)
{
    Spisok_1(); //Создали список из единственной структуры
    //Последовательно добавляем в список оставшиеся структуры
    for (int ij= 1; ij< iCh_Sotr; ij++)
        Vklyuchenie(&Konec, Sotrud[ij]);
} //SortClick
```

Список из трёх элементов приведен на рис 13.6.

По созданному списку структур можно последовательно перемещаться как вперёд, так и назад. Функция *VuvodClick* иллюстрирует возможность последовательного показа фамилий сотрудников при перемещении от первой до последней структуры списка.

```
void __fastcall TForm1::VuvodClick(TObject *Sender)
{
    Novuj= Nachalo;
    while (Novuj)
    {
        ShowMessage ((*Novuj).Sotr.Fam);
        Novuj= Novuj->Pravuj;
    } //while
} // VuvodClick
```

Поскольку указатель *Novuj* в начале тела функции направлен на первую структуру списка (у которой адрес не равен нулю или *NULL*), поэтому вход в цикл *while* разрешён. После обработки текущей структуры (вывода на экран фамилии сотрудника) в указатель *Novuj* записывается адрес структуры, располо

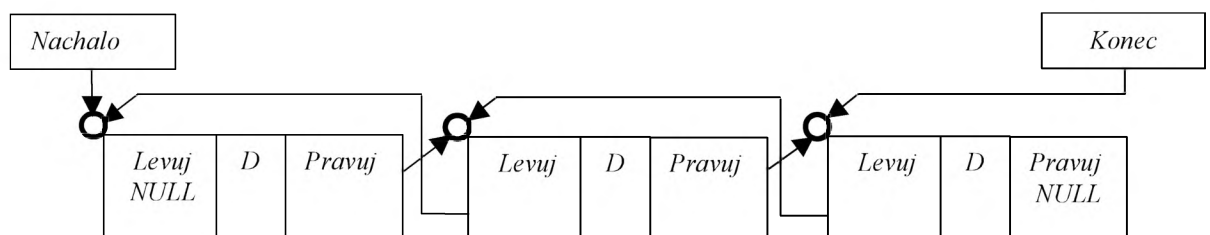


Рис. 13.6. Список из трёх структур

женной справа: $Novuj = Novuj \rightarrow Pravuj$; (см. рис. 13.6). В результате на экран выводится фамилия, записанная в следующей справа структуре списка. После просмотра последней структуры в указателе *Novuj* окажется *NULL*, скопированный (присвоенный) из поля *Pravuj* конечного элемента списка структур, что приведет к завершению работы цикла.

Функция *Vuvod_S_KoncaClick* осуществляет просмотр элементов прежнего списка структур из конца в начало. С этой целью в указатель *Novuj* помещается адрес конечного элемента: $Novuj = Konec$; а после обработки очередной структуры в указатель *Novuj* записывается адрес структуры, расположенной слева от текущей структуры: $Novuj = Novuj \rightarrow Levuj$. Работа цикла *while* завершается с переходом к первой структуре, у которой в поле *Levuj* хранится *NULL* (см. рис.13.6).

```
void __fastcall TForm1::Vuvod_S_KoncaClick(TObject *Sender)
{
    Novuj= Konec;
    while (Novuj)
    {
        ShowMessage ((*Novuj).Sotr.Fam);
        Novuj= Novuj->Levuj;
    } //while
} //Vuvod_S_KoncaClick
```

Если в поле *Pravuj* последней структуры записать адрес первой структуры, то получится кольцевой список. В этом списке при движении из начала в конец списка (при помощи функции *VuvodClick*) после просмотра последней структуры будет просматриваться первая структура: вы никогда при помощи кнопки с заголовком *Вывод на экран* (и с именем *Vuvod*) не выйдете из цикла просмотра (корректное завершение работы программы осуществляется командой *Ctrl+F2*). Операцию закольцовывания списка можно совершить разными способами, например, в конце функции *SozdatjSpisokClick*, добавить строку:

```
Konec->Pravuj= Nachalo;
```

Если в этой же функции добавить ещё и строку

```
(*Nachalo).Levuj= Konec;
```

то получится кольцевой список при движении от последнего элемента списка к первому (при помощи функции *Vuvod_S_KoncaClick*). В этом случае в поле *Levuj* первого элемента списка записывается адрес последней структуры, для корректного завершения программы воспользуйтесь упомянутой выше командой. В двух последних строках кода иллюстрируются *разные* способы доступа к полям структур.

Новые элементы списка можно вставлять не только в его конец, но и в любое заданное место в списке. Приведенная ниже функция *Alfavitnoe_Vklyuchenie* новые элементы списка вставляет таким образом, чтобы в итоговом списке структур фамилии сотрудников оказались упорядоченными по алфавиту. Ясно, что аналогичным образом можно сформировать упорядоченный список и по любому другому ключу – по любому полю с информацией об индивидуальных характеристиках сотрудника (например, по величине оклада). В эту функцию в качестве

параметра передаётся не только адрес ячейки памяти с именем *Konec_Spiska*, где хранится указатель – адрес *конечной* структуры списка, но и адрес указателя *Nachalo_Spiska* с адресом *начальной* структуры списка. Третий параметр функции *Alfavitnoe_Vklyuchenie* с именем *Sotrudnichek* выполняет ту же роль, что и одноимённый параметр в функции *Vklyuchenie*. Параметры *Nachalo_Spiska* и *Konec_Spiska* передаются в функцию по адресу (они являются указателями на указатель), поэтому адреса этих формальных параметров при вызове функции не изменяются. Объектная же часть упомянутых параметров-указателей может наполняться в ходе работы функции различными адресами структур формируемого упорядоченного списка структур.

В функции *Alfavitnoe_Vklyuchenie* предполагается, что список структур уже существует. Так же как и в функции *SozdatjSpisokClick*, список из единственной структуры подготавливается вызовом функции *Spisok_1()*. Однако о процессе создания всего списка будет рассказано ниже. Сейчас же сосредоточим внимание на разборе функции *Alfavitnoe_Vklyuchenie*: она для каждой новой структуры решает вопрос её местоположения в формируемом списке. Опишем вкратце её работу. Функция создаёт локальную переменную-структуру с именем *Novuj*. В её информационное поле *Sotr* помещаются данные об очередном сотруднике при помощи входного параметра *Sotrudnichek*. Далее создаётся *временная* переменная-указатель с именем *Yk*, куда записывается адрес начала списка:

```
Spis_Sotr *Yk= *Nachalo_Spiska;
```

Поскольку в переменной *Yk* теперь находится значение отличное от *NULL*, то вход в цикл разрешён, и он может просматривать в списке структуру за структурой. Как и ранее, для перехода от одной структуры к другой применяется операция:

```
Yk= Yk->Pravuj;
```

Для сравнения фамилий использована функция *AnsiCompareStr*. Ранее она в пособии не использовалась и не разъяснялась. Поэтому сообщаем, что эта функция выполняет сравнение двух своих параметров типа *String*. Если сравниваемые строки посимвольно эквивалентны, то функция возвращает нулевое значение. В случае же, когда при посимвольном сравнении какой-то символ первого параметра окажется меньше (его номер меньше в таблице кодов) или больше, чем соответствующий символ второго параметра, то функция возвращает значение меньше или соответственно больше нуля. Эту функцию применим для упорядочивания слов по алфавиту, она учитывает не только первые символы слова, но и все последующие.

В случае, когда исходный список состоит всего лишь из одного элемента, то функция *Alfavitnoe_Vklyuchenie* выясняет вопрос: поставить новую структуру слева или справа от существующей структуры. Поставить новую структуру справа от имеющейся единственной структуры списка означает вставить её в конец списка.

Если в результате предшествующих вызовов этой функции список имеет более одного элемента, то задача функции состоит в том, чтобы найти структуру, впереди которой по алфавиту следует поставить очередную новую структуру.

Однако если случится так, что все имеющиеся структуры уже просмотрены, а требуемое условие осталось нереализованным, то новая структура вставляется последней в список – после конечной структуры предшествующего списка.

При включении новой структуры в начало, внутрь или в конец списка следует побеспокоиться о связях элементов. Если структура включается в начало списка, то её полю *Levuji* следует назначить значение *NULL*, если в конец списка, то *NULL* назначается полю *Pravuij*. При этом в поле *Pravuij* начальной структуры записывается адрес последующей структуры (ранее она была начальной), а в поле *Levuji* последней структуры – адрес структуры, ставшей предпоследней. При вставке *внутри* списка в поле *Levuji* следует записать адрес предшествующей структуры, а в поле *Pravuij* – последующей структуры. Все эти операции аналогичны операциям, рассмотренным в функции *Vklyuchenie*, связи элементов списка поясняются рис. 13.6. Теперь прочтите код функции и его комментарии.

```
void Alfavitnoe_Vklyuchenie(Spis_Sotr **Nachalo_Spiska,
    Spis_Sotr **Konec_Spiska, strt_Sotr Sotrudnichek)
{
    //Добавляем структуру в список, локальная переменная
    Spis_Sotr *Novuij= new Spis_Sotr;
    //Записали информационную часть структуры
    Novuij->Sotr= Sotrudnichek;
    //Локальная вспомогательная переменная
    Spis_Sotr *Yk= *Nachalo_Spiska;
    //В Yk записали адрес начала списка структур
    //Просматриваем список с целью поиска места для очередной структуры
    //Список уже существует, поэтому вход в цикл разрешён
    while(Yk)
    {
        //Сравниваем строки с учётом регистра
        if(AnsiCompareStr(Sotrudnichek.Fam, Yk->Sotr.Fam)< 0)
        {
            //Заносим выбранную структуру перед текущим
            //элементом списка Yk. Для этого записываем в поле
            //Novuij->Pravuij адрес структуры Yk
            Novuij->Pravuij= Yk;
            //Если список состоит только из одной структуры и в
            //Yk записан её адрес
            if (Yk== *Nachalo_Spiska)
            {
                //Записываем новую структуру в начало списка
                //Обозначаем левую границу списка
                Novuij->Levuji= NULL;
                //В объект указателя Nachalo_Spiska записываем
                //адрес новой структуры
                *Nachalo_Spiska= Novuij;
            }
        }
        else //Вставляем между структурами списка
        {
            //Yk->Levuji- это адрес структуры, позади которой
            //вставляется структура Novuij
            //Записываем адрес структуры Novuij
            (Yk->Levuji)->Pravuij= Novuij;
            //Записываем адрес структуры, позади которой
            //вставляется структура Novuij
            Novuij->Levuji= Yk->Levuji;
        }
    }
}
```

```

        }//else_2
        //Завершение работы функции после вставки новой
        //структуры перед выбранной структурой в списке структур
        return;
    }//if_1
    //Перемещаемся по списку на одну структуру вправо
    Yk= Yk->Pravuj;
}//while
//Если при перемещении по списку обнаружен его конец, то
//цикл while игнорируется, а новую структуру следует
//вставить в конец списка
Novuj->Pravuj= NULL;
//(*Konec_Spiska) - Объект указателя конца списка
//с адресом последней структуры
Novuj->Levuuj= *Konec_Spiska;
//В поле Pravuj прежней последней структуры
//записываем адрес новой структуры
//Теперь бывшая последняя структура стала предпоследней
(*Konec_Spiska)->Pravuj= Novuj;
//В объект переменной Konec_Spiska записываем адрес
//новой (последней) структуры
*Konec_Spiska= Novuj;
}//Alfavitnoe_Vklyuchenie

```

Функция *Sort_SpisokClick* создаёт список структур, упорядоченный по алфавиту фамилий сотрудников. Эта функция при помощи вызова функции *Spisok_1()* создаёт исходный список из единственной структуры, а затем, с использованием цикла *for*, все структуры массива *Sotrud* последовательно вставляет в формируемый список. От функции *SozdatjSpisokClick* работа функция *Sort_SpisokClick* отличается лишь тем, что первая функция каждую текущую структуру массива *Sotrud* вставляет в конец формируемого списка, а последняя функция формирует список по алфавитному порядку сотрудников учреждения.

```

void __fastcall TForm1::Sort_SpisokClick(TObject *Sender)
{
    //Создаётся список из начальной структуры массива структур
    Spisok_1();
    //Последовательно в алфавитном порядке фамилий
    //добавляются в двухсвязный список оставшиеся структуры
    for (int ij= 1; ij< iCh_Sotr; ij++)
        Alfavitnoe_Vklyuchenie(&Nachalo, &Konec, Sotrud[ij]);
}
//Sort_SpisokClick

```

Все функции настоящего пункта следует добавить к копии программы, разработанной в п. 13.4.7. Новое приложение назовите *Двухсвязный список*, его интерфейс показан на рис 13.7. Командные кнопки с заголовками *Создать список*, *Вывод на экран*, *Вывод с конца списка*, *Отсортированный список* имеют соответственно имена: *SozdatjSpisok*, *Vuvod*, *Vuvod_S_Konca*, *Sort_Spisok*. Описанные выше функции с окончанием *Click*, являются функциями, обрабатывающими нажатие упомянутых командных кнопок.

В заключение укажем и на *недостаток* динамических структур данных. Он заключается в замедлении работы программы в связи с необходимостью выделения и освобождения памяти под динамические переменные. Эти процессы доста-

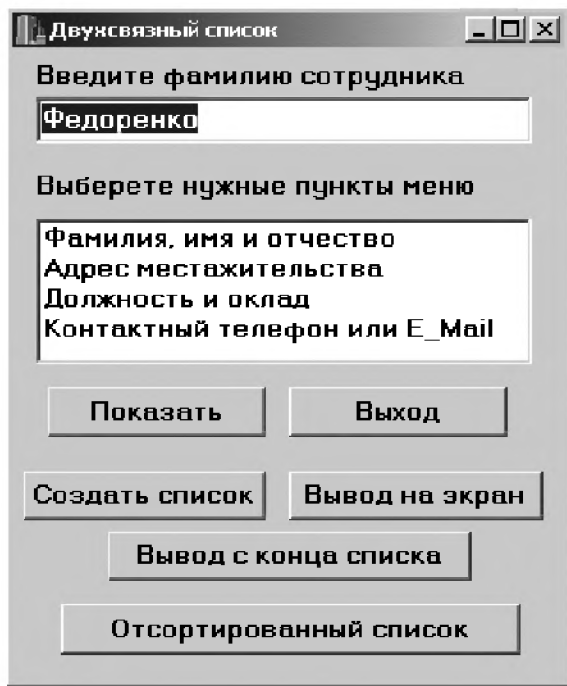


Рис. 13.7. Интерфейс приложения
Двухсвязный список

точно медленные. Поэтому, когда максимальный объем данных небольшой, эффективнее применять *статический* массив структур. В каждом конкретном случае программист *осознанно* выбирает, что более важно – быстродействие программы или объем оперативной памяти, занимаемый её данными. При быстродействии современных процессоров и объемах оперативной памяти для *очень большого* круга задач решение обсуждаемой проблемы становится *неактуальным*. Однако имеются и всегда будут иметься задачи, в которых приходится решать дилемму быстродействие-память.

Вопросы для самоконтроля

- ☐ Чему равен объем переменной перечислимого типа?
- ☐ Могут ли переменные-перечисления и сами константы-перечисления быть операндами математических выражений?
- ☐ Почему объем переменной типа множество *не зависит* от степени её заполнения?
- ☐ Какое *максимальное* количество элементов может содержаться во множестве?
- ☐ Назовите типы данных, применяемые в качестве элементов множеств.
- ☐ По какой причине буквы кириллицы неприменимы во множествах.
- ☐ Перечислите операции, разрешённые для однотипных множеств.
- ☐ Какой тип данных *недопустим* в объединениях?
- ☐ Зачем применяются вложенные структуры?
- ☐ Назовите тип поля, который неразрешён в структурах?
- ☐ Для чего используется наследование структур, защита их полей и методов?

- Как устроен динамический список структур? Укажите его достоинства и недостатки в сравнении со статическим массивом структур.

13.5. Задачи для программирования

Задача 13.1. Записать в текстовый файл *Baza_1.txt* сведения об автомобилях: номер; тип машины; фамилия, имя, отчество и адрес владельца. Программа в файл *Rezyltat.txt* должна вывести сведения об автомобилях, номера которых содержат три заданные цифры в любом порядке. Задачу решите с использованием вложенных структур.

Задача 13.2. Итоги сессии группы студентов запишите в файл *Baza_2.txt*. Программа в файл *Rezyltat.txt* должна включить содержимое исходного файла и списки студентов с заголовками: *отличники*, *хорошисты*, *неуспевающие*.

Задача 13.3. В файл *Baza_3.txt* запишите сведения о работниках завода: фамилию, имя, отчество; дату рождения; должность; число месяцев невыплаченной зарплаты за прошлый год, пол. Программа в файл *Rezyltat.txt* должна вывести список работников-пенсионеров, у которых число неоплаченных месяцев работы за прошлый год превышает 6 месяцев.

13.6. Варианты решений задач

Решения задачи 13.1

Интерфейс приложения по задаче 13.1 имеет заголовок *Автомобили сотрудников* и показан на рис 13.8.

Поля ввода и командные кнопки имеют имена: *Vvod_Cif_1*, *Vvod_Cif_2*, *Vvod_Cif_3*, *Poisk*, *Vuxod*. Параметры полей базы данных, приведенной в файле, показаны в следующей таблице.

Имя поля	Позиция начала	Длина поля
Номер автомобиля	1	9
Тип автомобиля	10	10
Фамилия, имя, отчество	20	32
Индекс	52	7
Город	59	11
Улица	70	15
Номер дома	85	6
Номер квартиры	91	3

Код приложения *Автомобили сотрудников* показан ниже.

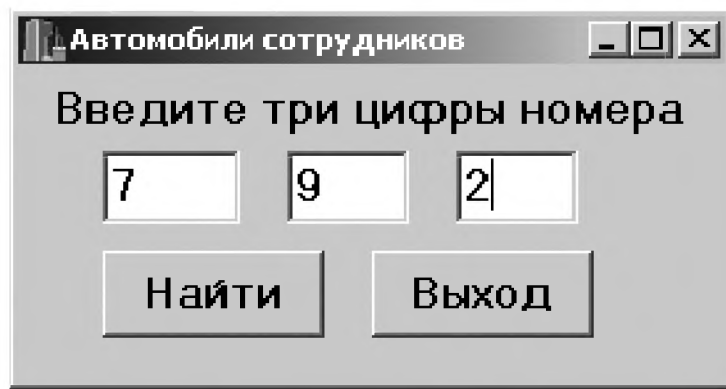


Рис. 13.8. Интерфейс приложения Автомобили сотрудников

Код решения задачи 13.1

```
//Ориентировочное число автомобилистов
const int iChisloAvtomoilistov= 30;
//Фактическое число автомобилистов в базе данных
int iChislo_Avtomoilistov;
//Для ввода данных из файла и вывода данных в файл
TStringList *Spisok_Isx= new TStringList,
               *Spisok_Rez= new TStringList;
//Файловые переменные для файлов с данными и результатами
String Isx_f= "Baza_1.txt", Rez_f= "Rezyltat.txt";

void Vvod_Strok_Spiska(int & iChislo_Avtomoilistov)
{
    try
    {
        Spisok_Isx->LoadFromFile(Isx_f);
    }//try
    catch (...)
    {
        ShowMessage("Файл \" " + Isx_f + " \" не найден");
        Abort();
    }//catch
    iChislo_Avtomoilistov= Spisok_Isx->Count;
}

struct struct_Adres
{
    unsigned long Indeks;
    String Gorod, Ylica, Nomer_Doma;
    short Nomer_Kvartiru;
};

struct struct_Avtomobil
{
    String Nomer,
           Tip;
};

struct struct_AvtoLyubitel
{
    String Familia_Imla_Otchestvo;
    struct_Adres Adres;
    struct_Avtomobil Avtomobil;
```

```

} AvtoLyubiteli[iChisloAvtomoilistov]; //Массив структур
void Zapolnenie_Stryktyru(int iChislo_Avtomoilistov)
{
    for (int ij= 0; ij< iChislo_Avtomoilistov; ij++)
    {
        String Stroka= Spisok_Isx->Strings[ij];
        AvtoLyubiteli[ij].Avtomobil.Nomer=
            Trim(Stroka.SubString(1, 9));
        String Stroka_1= Stroka.SubString(10, 10);
        AvtoLyubiteli[ij].Avtomobil.Tip= Stroka_1.Trim();
        AvtoLyubiteli[ij].Familiya_Imja_Otchestvo=
            Trim(Stroka.SubString(20, 32));
        Stroka_1= Stroka.SubString(52, 7);
        AvtoLyubiteli[ij].Adres.Indeks=
            StrToInt(Stroka_1.Trim());
        AvtoLyubiteli[ij].Adres.Gorod=
            Trim(Stroka.SubString(59, 11));
        AvtoLyubiteli[ij].Adres.Ylica=
            Trim(Stroka.SubString(70, 15));
        AvtoLyubiteli[ij].Adres.Nomer_Doma=
            Trim(Stroka.SubString(85, 6));
        Stroka_1= Stroka.SubString(91, 3);
        AvtoLyubiteli[ij].Adres.Nomer_Kvartiru=
            StrToInt(Stroka_1.Trim());
    } //for_ij
} //Zapolnenie_Stryktyru

void __fastcall TAvtomobili::FormCreate(TObject *Sender)
{
    //Данные базы переписываем из файла в переменную Spisok_Isx
    //Подсчёт фактического числа строк базы данных
    Vvod_Strok_Spiska(iChislo_Avtomoilistov);
    //Заполняем поля структуры при помощи переменной Spisok_Isx
    Zapolnenie_Stryktyru(iChislo_Avtomoilistov);
} //FormCreate

//Поиск номера автомобиля, у которого имеются три заданные
//цифры в любом порядке
void __fastcall TAvtomobili::PoiskClick(TObject *Sender)
{
    Spisok_Rez->Clear();
    Spisok_Rez->Add("Данные об искомых автомобилях");
    Spisok_Rez->Add(""); //Пропуск строки
    bool bPoisk= false; //Индикатор найденного номера
    for (int ij= 0; ij< Spisok_Isx->Count; ij++)
    {
        String Strora = AvtoLyubiteli[ij].Avtomobil.Nomer;
        if (Strora.Pos(Avtomobili->Vvod_Cif_1->Text) &&
            Strora.Pos(Avtomobili->Vvod_Cif_2->Text) &&
            Strora.Pos(Avtomobili->Vvod_Cif_3->Text) &&
            // Цифры должны находиться в разных позициях
            Strora.Pos(Avtomobili->Vvod_Cif_1->Text) !=
                Strora.Pos(Avtomobili->Vvod_Cif_2->Text) &&
            Strora.Pos(Avtomobili->Vvod_Cif_1->Text) !=
                Strora.Pos(Avtomobili->Vvod_Cif_3->Text) &&
            Strora.Pos(Avtomobili->Vvod_Cif_2->Text) !=
                Strora.Pos(Avtomobili->Vvod_Cif_3->Text))
        {
            //Найден номер
            Strora = Strora.SubString(1, 9);
            Strora = Strora.SubString(10, 10);
            Strora = Strora.SubString(20, 32);
            Strora = Strora.SubString(52, 7);
            Strora = Strora.SubString(59, 11);
            Strora = Strora.SubString(70, 15);
            Strora = Strora.SubString(85, 6);
            Strora = Strora.SubString(91, 3);
            Strora = Strora.SubString(100, 10);
            Strora = Strora.SubString(110, 10);
            Strora = Strora.SubString(120, 10);
            Strora = Strora.SubString(130, 10);
            Strora = Strora.SubString(140, 10);
            Strora = Strora.SubString(150, 10);
            Strora = Strora.SubString(160, 10);
            Strora = Strora.SubString(170, 10);
            Strora = Strora.SubString(180, 10);
            Strora = Strora.SubString(190, 10);
            Strora = Strora.SubString(200, 10);
            Strora = Strora.SubString(210, 10);
            Strora = Strora.SubString(220, 10);
            Strora = Strora.SubString(230, 10);
            Strora = Strora.SubString(240, 10);
            Strora = Strora.SubString(250, 10);
            Strora = Strora.SubString(260, 10);
            Strora = Strora.SubString(270, 10);
            Strora = Strora.SubString(280, 10);
            Strora = Strora.SubString(290, 10);
            Strora = Strora.SubString(300, 10);
            Strora = Strora.SubString(310, 10);
            Strora = Strora.SubString(320, 10);
            Strora = Strora.SubString(330, 10);
            Strora = Strora.SubString(340, 10);
            Strora = Strora.SubString(350, 10);
            Strora = Strora.SubString(360, 10);
            Strora = Strora.SubString(370, 10);
            Strora = Strora.SubString(380, 10);
            Strora = Strora.SubString(390, 10);
            Strora = Strora.SubString(400, 10);
            Strora = Strora.SubString(410, 10);
            Strora = Strora.SubString(420, 10);
            Strora = Strora.SubString(430, 10);
            Strora = Strora.SubString(440, 10);
            Strora = Strora.SubString(450, 10);
            Strora = Strora.SubString(460, 10);
            Strora = Strora.SubString(470, 10);
            Strora = Strora.SubString(480, 10);
            Strora = Strora.SubString(490, 10);
            Strora = Strora.SubString(500, 10);
            Strora = Strora.SubString(510, 10);
            Strora = Strora.SubString(520, 10);
            Strora = Strora.SubString(530, 10);
            Strora = Strora.SubString(540, 10);
            Strora = Strora.SubString(550, 10);
            Strora = Strora.SubString(560, 10);
            Strora = Strora.SubString(570, 10);
            Strora = Strora.SubString(580, 10);
            Strora = Strora.SubString(590, 10);
            Strora = Strora.SubString(600, 10);
            Strora = Strora.SubString(610, 10);
            Strora = Strora.SubString(620, 10);
            Strora = Strora.SubString(630, 10);
            Strora = Strora.SubString(640, 10);
            Strora = Strora.SubString(650, 10);
            Strora = Strora.SubString(660, 10);
            Strora = Strora.SubString(670, 10);
            Strora = Strora.SubString(680, 10);
            Strora = Strora.SubString(690, 10);
            Strora = Strora.SubString(700, 10);
            Strora = Strora.SubString(710, 10);
            Strora = Strora.SubString(720, 10);
            Strora = Strora.SubString(730, 10);
            Strora = Strora.SubString(740, 10);
            Strora = Strora.SubString(750, 10);
            Strora = Strora.SubString(760, 10);
            Strora = Strora.SubString(770, 10);
            Strora = Strora.SubString(780, 10);
            Strora = Strora.SubString(790, 10);
            Strora = Strora.SubString(800, 10);
            Strora = Strora.SubString(810, 10);
            Strora = Strora.SubString(820, 10);
            Strora = Strora.SubString(830, 10);
            Strora = Strora.SubString(840, 10);
            Strora = Strora.SubString(850, 10);
            Strora = Strora.SubString(860, 10);
            Strora = Strora.SubString(870, 10);
            Strora = Strora.SubString(880, 10);
            Strora = Strora.SubString(890, 10);
            Strora = Strora.SubString(900, 10);
            Strora = Strora.SubString(910, 10);
            Strora = Strora.SubString(920, 10);
            Strora = Strora.SubString(930, 10);
            Strora = Strora.SubString(940, 10);
            Strora = Strora.SubString(950, 10);
            Strora = Strora.SubString(960, 10);
            Strora = Strora.SubString(970, 10);
            Strora = Strora.SubString(980, 10);
            Strora = Strora.SubString(990, 10);
            Strora = Strora.SubString(1000, 10);
            Strora = Strora.SubString(1010, 10);
            Strora = Strora.SubString(1020, 10);
            Strora = Strora.SubString(1030, 10);
            Strora = Strora.SubString(1040, 10);
            Strora = Strora.SubString(1050, 10);
            Strora = Strora.SubString(1060, 10);
            Strora = Strora.SubString(1070, 10);
            Strora = Strora.SubString(1080, 10);
            Strora = Strora.SubString(1090, 10);
            Strora = Strora.SubString(1100, 10);
            Strora = Strora.SubString(1110, 10);
            Strora = Strora.SubString(1120, 10);
            Strora = Strora.SubString(1130, 10);
            Strora = Strora.SubString(1140, 10);
            Strora = Strora.SubString(1150, 10);
            Strora = Strora.SubString(1160, 10);
            Strora = Strora.SubString(1170, 10);
            Strora = Strora.SubString(1180, 10);
            Strora = Strora.SubString(1190, 10);
            Strora = Strora.SubString(1200, 10);
            Strora = Strora.SubString(1210, 10);
            Strora = Strora.SubString(1220, 10);
            Strora = Strora.SubString(1230, 10);
            Strora = Strora.SubString(1240, 10);
            Strora = Strora.SubString(1250, 10);
            Strora = Strora.SubString(1260, 10);
            Strora = Strora.SubString(1270, 10);
            Strora = Strora.SubString(1280, 10);
            Strora = Strora.SubString(1290, 10);
            Strora = Strora.SubString(1300, 10);
            Strora = Strora.SubString(1310, 10);
            Strora = Strora.SubString(1320, 10);
            Strora = Strora.SubString(1330, 10);
            Strora = Strora.SubString(1340, 10);
            Strora = Strora.SubString(1350, 10);
            Strora = Strora.SubString(1360, 10);
            Strora = Strora.SubString(1370, 10);
            Strora = Strora.SubString(1380, 10);
            Strora = Strora.SubString(1390, 10);
            Strora = Strora.SubString(1400, 10);
            Strora = Strora.SubString(1410, 10);
            Strora = Strora.SubString(1420, 10);
            Strora = Strora.SubString(1430, 10);
            Strora = Strora.SubString(1440, 10);
            Strora = Strora.SubString(1450, 10);
            Strora = Strora.SubString(1460, 10);
            Strora = Strora.SubString(1470, 10);
            Strora = Strora.SubString(1480, 10);
            Strora = Strora.SubString(1490, 10);
            Strora = Strora.SubString(1500, 10);
            Strora = Strora.SubString(1510, 10);
            Strora = Strora.SubString(1520, 10);
            Strora = Strora.SubString(1530, 10);
            Strora = Strora.SubString(1540, 10);
            Strora = Strora.SubString(1550, 10);
            Strora = Strora.SubString(1560, 10);
            Strora = Strora.SubString(1570, 10);
            Strora = Strora.SubString(1580, 10);
            Strora = Strora.SubString(1590, 10);
            Strora = Strora.SubString(1600, 10);
            Strora = Strora.SubString(1610, 10);
            Strora = Strora.SubString(1620, 10);
            Strora = Strora.SubString(1630, 10);
            Strora = Strora.SubString(1640, 10);
            Strora = Strora.SubString(1650, 10);
            Strora = Strora.SubString(1660, 10);
            Strora = Strora.SubString(1670, 10);
            Strora = Strora.SubString(1680, 10);
            Strora = Strora.SubString(1690, 10);
            Strora = Strora.SubString(1700, 10);
            Strora = Strora.SubString(1710, 10);
            Strora = Strora.SubString(1720, 10);
            Strora = Str
```

```

        Strora.Pos(Avtomobili->Vvod_Cif_3->Text))
    {
        Spisok_Rez->Add(AvtoLyubiteli[ij].Avtomobil.Nomer +
            ", " + AvtoLyubiteli[ij].Avtomobil.Tip + ", " +
            AvtoLyubiteli[ij].Familiya_Imja_Otchestvo + ", " +
            IntToStr(AvtoLyubiteli[ij].Adres.Indeks) + ", г. " +
            AvtoLyubiteli[ij].Adres.Gorod + ", ул. " +
            AvtoLyubiteli[ij].Adres.Ylica + ", д. " +
            AvtoLyubiteli[ij].Adres.Nomer_Doma + ", кв. " +
            IntToStr(AvtoLyubiteli[ij].Adres.Nomer_Kvartiru));
        bPoisk= true;
    } // if
} // for
if (bPoisk== false) //Если ни один номер не найден
    ShowMessage(«Номера с таким набором цифр отсутствуют в базе данных»);
Spisok_Rez->SaveToFile(Rez_f);
} // PoiskClick

void __fastcall TAvtomobili::VuxodClick(TObject *Sender)
{
    delete Spisok_Isx;
    delete Spisok_Rez;
    Close();
} // VuxodClick

```

Решение задачи 13.2

Содержимое файла *Baza_2.txt* для задачи 13.2:

Шевченко	5	5	5	5	5
Пушкин	3	3	3	3	3
Серов	4	5	4	5	5
Федоренко	3	5	2	4	2

Код решения задачи 13.2

```

//Ориентировочное число студентов
const int iChisloStydentov= 30,
        iChislo_Disciplin= 5; //Число дисциплин
//Фактическое число студентов в базе данных
int iChislo_Stydentov;
//Индикатор принадлежности студента к заданной подгруппе
bool Rezyltatu_Testa;
//Для ввода и вывода данных из файла
TStringList *Spisok_Isx= new TStringList,
            *Spisok_Rez= new TStringList;
//Файловые переменные для файлов с данными и результатами
String Isx_f= "Baza_2.txt",
        Rez_f= "Rezyltat.txt";
//Массив оценок для каждого студента
typedef char Massiv_Ocenok[iChislo_Disciplin];

void Vvod_Strok_Spiska(int & iChislo_Stydentov)
{
    try
    {
        Spisok_Isx->LoadFromFile(Isx_f);
    }
}

```

```
    }//try
    catch (...)
    {
        ShowMessage("Файл \" " + Isx_f + " \" не найден");
        Abort();
    }//catch
    iChislo_Stydentov= Spisok_Isx->Count;
    }//Vvod_Strok_Spiska

    struct struct_ItoGi_Sessii
    {
        String Familija;
        Massiv_Ocenok Mas_Ocenok;
    } Stydentu[iChisloStydentov]; //Массив структур

void Zapolnenie_Stryktyru(int iChislo_Stydentov)
{
    for (int ij= 0; ij< iChislo_Stydentov; ij++)
    {
        String Stroka= Spisok_Isx->Strings[ij];
        Stydentu[ij].Familija= Trim(Stroka.SubString(1, 14));
        int Pos_Ocenki= 15;
        for (int iDiscip= 0; iDiscip< iChislo_Disciplin; iDiscip++)
        {
            String Stroka_1= Stroka.SubString(Pos_Ocenki, 1);
            Stydentu[ij].Mas_Ocenok[iDiscip]= Stroka_1[1];
            Pos_Ocenki= Pos_Ocenki + 2;
        }//for_iDiscip
    }//for_ij
} //Zapolnenie_Stryktyru

void __fastcall TYspev_Stydentov::VuxodClick(TObject *Sender)
{
    delete Spisok_Isx;
    delete Spisok_Rez;
    Close();
} //VuxodClick

void __fastcall TYspev_Stydentov::FormCreate(TObject *Sender)
{
    //Данные базы переписываем из файла в переменную Spisok_Isx
    //Подсчёт фактического числа строк базы данных
    Vvod_Strok_Spiska(iChislo_Stydentov);
    //Заполняем поля структуры при помощи переменной Spisok_Isx
    Zapolnenie_Stryktyru(iChislo_Stydentov);
} //FormCreate

//Распечатывает фамилии студентов с минимальной оценкой
//Ocenka. Если таких студентов нет, то Rez_Testa= false
void Obrabotka(bool &Rez_Testa, char Ocenka)
{
    Rez_Testa= true;
    //Счётчик числа студентов, у которых Min_Ocenka!= Ocenka
    int iS= 0;
    for (int ij= 0; ij< iChislo_Stydentov; ij++)
    {
        char Min_Ocenka= Stydentu[ij].Mas_Ocenok[0];
```

```

    for (int iNom_Discipl= 0; iNom_Discipl<
            iChislo_Disciplin; iNom_Discipl++)
        //Поиск минимальной оценки студента
        if (Stydentu[ij].Mas_Ocenok[iNom_Discipl]< Min_Ocenka)
            Min_Ocenka= Stydentu[ij].Mas_Ocenok[iNom_Discipl];
    if (Min_Ocenka== Ocenka)
        Spisok_Rez->Add(Stydentu[ij].Familiya);
    else
        iS++;
} //for_ij
if (iS== iChislo_Stydentov)
    Rez_Testa= false;
} //Obrabotka

//Выполняет решение задачи
void __fastcall Tyspev_Stydentov::RaspredelenieClick(TObject *Sender)
{
    Spisok_Rez->Clear();
    Spisok_Rez->Add("Исходная ведомость");
    Spisok_Rez->Add(""); //Пропуск строки
    for (int ij= 0; ij< iChislo_Stydentov; ij++)
        Spisok_Rez->Add(Spisok_Isx->Strings[ij]);
    Spisok_Rez->Add(""); //Пропуск строки
    Spisok_Rez->Add("Успеваемость группы студентов");
    for (char iOcenka= '5'; iOcenka>= '2'; iOcenka-)
    {
        switch (iOcenka)
        {
            case '5': Spisok_Rez->Add("Отличники");
                       break;
            case '4': Spisok_Rez->Add("Хорошисты");
                       break;
            case '3': Spisok_Rez->Add("Троечники");
                       break;
            case '2': Spisok_Rez->Add("Неуспевающие");
        } //switch
        Obrabotka(Rezyltatu_Testa, iOcenka);
        if (!Rezyltatu_Testa)
            Spisok_Rez->Add("Отсутствуют");
        Spisok_Rez->Add("");
    } //for_iOcenka
    Spisok_Rez->SaveToFile(Rez_f);
} //RaspredelenieClick

```

Решения задачи 13.3

Параметры полей базы данных, приведенной в файле *Baza_3.txt*.

Имя поля	Позиция начала	Длина поля
Фамилия, имя, отчество	1	32
Дата рождения (например, 10.11.1949)	33	12
Должность	45	12
Число месяцев невыплат зарплаты	57	2
Пол работника (м или ж)	59	1

Код решения задачи 13.3

```
//Ориентировочное число работников
const int iChisloRabotnikov= 30,
        //Предельный период невыплат в месяцах
        iChislo_Mes_Nevuplat= 6;
//Фактическое число работников в базе данных
int iChislo_Rabotnikov;
//Для ввода данных из файла и вывода данных в файл
TStringList *Spisok_Isx= new TStringList,
        *Spisok_Rez= new TStringList;
//Файловые переменные для файлов с данными и результатами
String Isx_f= "Baza_3.txt",
        Rez_f= "Rezyltat.txt";

void Vvod_Strok_Spiska(int & iChislo_Rabotnikov)
{
    try
    {
        Spisok_Isx->LoadFromFile(Isx_f);
    }
    catch (...)
    {
        ShowMessage("Файл \" " + Isx_f + " \" не найден");
        Abort();
    }
    iChislo_Rabotnikov= Spisok_Isx->Count;
} //Vvod_Strok_Spiska

struct struct_Rabotnik
{
    String Familia_Imlja_Otchestvo,
        Data_Rogden,
        Dolgnostj;
    short Chislo_Mes; //Число месяцев невыплаты зарплаты
    char Pol; //Пол
} Rabotniki[iChisloRabotnikov]; //Массив структур

void Zapolnenie_Stryktyru(int iChislo_Rabotn)
{
    String Stroka;
    for (int ij= 0; ij< iChislo_Rabotn; ij++)
    {
        String Stroka= Spisok_Isx->Strings[ij];

        Rabotniki[ij].Familija_Imlja_Otchestvo=
            Trim(Stroka.SubString(1, 32));
        String Stroka_1= Stroka.SubString(33, 12);
        Rabotniki[ij].Data_Rogden= Stroka_1.Trim();
        Rabotniki[ij].Dolgnostj= Trim(Stroka.SubString(45,12));
        Rabotniki[ij].Chislo_Mes=
            StrToInt(Trim(Stroka.SubString(57, 2)));
        Stroka_1= Stroka.SubString(59, 1);
        Rabotniki[ij].Pol= Stroka_1[1];
    } //for_int
} //Zapolnenie_Stryktyru
```

```
//Определяет возраст в днях по дате рождения
unsigned long Chislo_Dnej(String Data)
{
    unsigned long x_Den, x_Mes, x_God, Chislo_Dn;
    x_Den= StrToInt(Data.SubString(1, 2));
    x_Mes= StrToInt(Data.SubString(4, 2));
    x_God= StrToInt(Trim(Data.SubString(7, 4)));
    Chislo_Dn= x_Den + x_Mes*30 + x_God*365;
    return Chislo_Dn;
}

unsigned long Chislo_Dnej_Segodnja(void)
{
    //В переменную T передаются текущие дата и время
    TDateTime T(Now());
    //Текущая дата, например, 18.08.04
    String Data_Tekysch= T.DateString();
    unsigned long Chislo_Dnej_Segodn=
        Chislo_Dnej(Data_Tekysch)+ 2000*365;
    return Chislo_Dnej_Segodn;
}

void __fastcall TPensioneru::VuxodClick(TObject *Sender)
{
    delete Spisok_Isx;
    delete Spisok_Rez;
    Close();
}

void __fastcall TPensioneru::PokazatjClick(TObject *Sender)
{
    Spisok_Rez->Add("Работающие пенсионеры, которым не\
        выплачивалась зарплата более " +
        IntToStr(iChislo_Mes_Nevuplat) + " месяцев:");
    int iNom= 0; //Счётчик числа искомых работников-пенсионеров
    for (int ij= 0; ij< iChislo_Rabotnikov; ij++)
    {
        int Pension_Vozrast, Vozrast;
        if (Rabotniki[ij].Pol== 'ж')
            Pension_Vozrast= 55;
        else
            Pension_Vozrast= 60;
        //Возраст работника или работницы
        Vozrast= (Chislo_Dnej_Segodnja() -
            Chislo_Dnej(Rabotniki[ij].Data_Rogden))/365;
        if ((Vozrast>= Pension_Vozrast) &&
            (Rabotniki[ij].Chislo_Mes> iChislo_Mes_Nevuplat))
        {
            iNom++;
            String Stroka= IntToStr(iNom++) + " " +
                Rabotniki[ij].Familija_Imja_Otchestvo;
            Spisok_Rez->Add(Stroka);
        }
    }
    if (iNom== 0)
        Spisok_Rez->Add("Искомые работники не обнаружены");
}
```



```
Spisok_Rez->SaveToFile(Rez_f);  
} //PokazatjClick
```

```
void __fastcall TPensioneru::FormCreate(TObject *Sender)  
{  
    //Данные базы переписываем из файла в переменную Spisok_Isx  
    Vvod_Strok_Spiska(iChislo_Rabotnikov);  
    //Заполняем поля структуры при помощи переменной Spisok_Isx  
    //Подсчёт фактического числа строк базы данных  
    Zapolnenie_Stryktyru(iChislo_Rabotnikov);  
} //FormCreate
```



Урок 14. Классы

14.1. Вводные замечания

Весь *предшествующий* материал учебника можно считать *лишь подготовкой* для рассмотрения вопросов *настоящего* урока, посвящённого классам. Вместе с тем о классах упоминалось на всех уроках, ведь изучаемый язык визуального программирования *основывается* на таком типе данных. С этого урока пользовательские классы станут основой наших учебных программ. Однако вначале *подытожим* и обобщим всё то, что уже сообщалось о классах в нашей книге.

Прежде всего, отметим, что *практически все* возможности объединений и структур *полностью* относятся и к классам, поскольку перечисленные типы данных представляют собой *частные случаи* классов. Все *различия* между этими типами перечисляются в конце урока (см. раздел 14.13). Сейчас же остановимся только на двух отличиях между классами и структурами. *Первое* из них состоит в том, что в структурах *по умолчанию* используется спецификатор доступа *public* (открытый, общедоступный), а в классах – *private* (закрытый, собственный). С учётом этого различия в приложениях для замены структур классами необходимо:

- ❑ ключевые слова *struct* заместить ключевыми словами *class*;
- ❑ в начале описания общедоступных полей и методов поставить служебное слово *public* (с двоеточием в конце);
- ❑ если класс наследует один или более базовых классов (множественное наследование), то слово *public* (без двоеточия) разместить впереди *каждо-го* имени класса предка.

Второе различие между классами и структурами заключается в следующем: помимо, используемых в структурах спецификаторов доступа *public*, *private* и *protected* (защищённый), в классах *C++Builder* применяется раздел *__published* (объявленный, опубликованный). К этому разделу прибегают в случаях, когда пользовательский класс наследует компоненты библиотеки визуальных компонентов *VCL* и требуется, чтобы поля-данные были доступны не только программно, но и в *Инспекторе Объектов* (в ходе проектирования приложения). При этом доступ к полям обеспечивается при помощи свойств чтения и записи данных в поля. Эти свойства реализуются методами чтения и записи, расположенными в классах визуальных компонентов.

С визуальными объектами и их наследниками могут происходить различные

события (например, *OnClick*). Событие – это специальное свойство, являющееся обобщённым указателем функции. Обычно таким указателем является указатель *this* (о нём рассказывается ниже). Тело функции по обработке того или иного события разрабатывается программистом. В настоящем пособии вопросы модернизации и расширения компонентов *VCL* не рассматриваются.

Сейчас же перечислим другие понятия и процессы, перешедшие из структур в классы. Классы *содержат* в себе (или инкапсулируют) поля и методы для обработки полей-данных. Классы разрешается вкладывать друг в друга. Классы имеют возможность наследовать поля и методы своего предка или предков. При этом методы могут *переопределяться* в каждом дочернем классе: использоваться с тем же именем, что и в родительском классе, однако с другим телом и параметрами в заголовке. При изучении структур последняя возможность не упоминалась, поэтому подробно проиллюстрируем её в настоящем уроке на примере классов. Методы класса имеют доступ к любым другим методам и любым полям этого же класса, как открытым, так защищённым и закрытым. Переопределять можно *только методы*, переопределение полей запрещено: имена полей всех классов-наследников должны быть уникальными – их идентификаторы не могут повторяться ни в базовом классе, ни в дочерних классах. Поэтому после объявления поля данных *никакой дочерний класс* не может объявить поле данных с таким же именем.

Далее остановимся на *особенностях* классов. Вложение классов *на практике* обычно не применяется (его заменяет наследование), поля классов рекомендуется, по возможности, делать закрытыми. Согласно утверждениям профессионалов, в редких случаях представляется полезным, чтобы поля класса были защищёнными для осуществления возможности их обработки и методами наследуемых классов.

Объектно-ориентированное программирование (ООП) основывается на трёх принципах: *инкапсуляция, наследование и полиморфизм*. Последнее понятие рассмотрим в разделе 14.4, первые два – разъясним сейчас.

14.2. Инкапсуляция и наследование

Инкапсуляция или *скрытие* данных и методов их обработки в едином блоке направлена на повышение надёжности программ, сокращение сроков их разработки. Зачатки инкапсуляции наблюдаются уже в случае применения пользовательских функций, когда за именем функции скрываются определённые операции. Эти операции или действия можно выполнять *многократно* с одними и теми же или различными параметрами. При этом любую операцию имеется возможность реализовать по-разному, разнообразными способами (различными вариантами функции). При вызове конкретной функции следует принимать во внимание лишь её назначение, количество параметров, их последовательность и тип. Конкретная функция может вызываться в различных программных модулях, использоваться в *разных* программах.

Применение классов повышает надёжность приложения, поскольку в них

(аналогично объединениям и структурам) осуществляется *более высокая* степень инкапсуляции, в результате чего данные обрабатываются, как правило, с использованием предназначенных для них методов.

Сократить размер программы, повысить надёжность и ускорить процесс её разработки помимо инкапсуляции позволяет также и наследование. В ходе наследования возможности одного типа (объединений, структур и классов) полностью или частично переходят к другому типу, что позволяет без каких-либо изменений использовать разработанные ранее фрагменты кода (что и приводит к уменьшению размера программы и времени её разработки).

Далее детально рассматриваются классы, их устройство и возможности. Ещё раз напоминаем, что класс это *тип данных*, в котором имеются *поля* и *методы*. Поля в самом классе (точно также как и в структурах) инициализировать *не решается*. При объявлении переменной заданного класса говорят, что порождается *объект* или *экземпляр класса* – это блок ячеек в оперативной памяти компьютера, в котором размещаются *только поля*, заявленные при объявлении класса. Методы же класса в объекты *не копируются*, они в *единственном* экземпляре содержатся *только* в классе (также как и в структуре).

Для передачи конкретного поля выбранного объекта заданному методу класса используется скрытый параметр-поле объекта с именем *this*. Он является указателем и *автоматически* настраивается на адрес объекта, в котором сам расположен (т. е. в него записывается адрес объекта). Более подробно о неявном параметре *this* рассказывается в разделе 14.10.

Для разъяснения как упомянутых выше, так и других возможностей классов рассмотрим упрощённую задачу. На её примере механизмы работы с классами, скорее всего, покажутся весьма надуманными и очень уж неуклюжими. Однако не следует забывать о том, что ООП предназначено (и особенно эффективно) для применения в *очень больших* программах, когда число строк кода значительно больше 10 000. ООП является *вершиной* технологии программирования, к которой разработчики языков высокого уровня шли на протяжении *многих* лет. Поэтому противопоставление ООП *иных* (прежних) способов разработки программ просто неуместно.

Итак, условие учебной задачи. Разработаем программу для расчёта единой платёжной ведомости высшего учебного заведения, в которой присутствуют студенты, сотрудники и преподаватели. У каждой из перечисленных категорий сумма ежемесячных выплат определяется по-разному: у студентов – на основе среднего балла последней сессии, у сотрудников – окладом и премией, а у преподавателей – окладом, премией и величиной почасовой нагрузки (числом прочитанных часов и стоимостью часа). Каждая строка ведомости начинается фамилией, именем и отчеством людей из перечисленных групп (отнеситесь снисходительно к определённой искусственности этой задачи).

Конечно, в реальной программе для расчёта такой и подобных ведомостей должна иметься база данных, обычно располагаемая в текстовом файле. Для каждого человека она содержит фамилию, имя и отчество, категорию (студент, со-

трудник, преподаватель), данные, необходимые для начисления выплат конкретному лицу. Однако в таком *реальном* варианте существенно увеличивается код программы и поэтому теряется наглядность объяснения *новых* понятий. Вместе с тем все операции по перезаписи данных базы по полям массива структур неоднократно иллюстрировались на прошлом уроке, что даёт возможность читателям (при необходимости) осуществить аналогичные преобразования данных и в случае с экземплярами классов (все шаги по замене структур классами перечислены в начале раздела 14.1).

В целях сокращения размера учебного приложения и улучшения ясности изложения, инициализировать поля экземпляров классов будем вводом необходимых данных *в самой* программе (в функции *FormCreate*), а список в платёжной ведомости ограничим только *одним* представителем из каждой упомянутой категории лиц.

Ниже приведены фрагменты кода этой программы. Её подробный анализ даётся в конце кода. Сейчас же ограничимся лишь отдельными замечаниями. Как уже не раз отмечалось на предшествующих уроках, описание классов и реализации их методов следует размещать в *разных* программных модулях. Это способствует ускорению модернизации приложений и в ООП является определённым стандартом. Данное приложение задумано небольшим, что позволяет нам в целях удобства изложения не создавать *новые* модули: описание пользовательских классов разместим в файле, в котором описывается класс формы (заголовочный файл главной формы), а реализацию методов поместим в основной файл реализации проекта.

Напоминаем: для того чтобы попасть из *Редактора Кода* в заголовочный файл формы следует ввести команду *Ctrl+F6* или команду контекстного меню *Open Source/Header File*, можно также воспользоваться закладкой или страницей (с расширением *h*), расположенной внизу окна *Редактора кода* (только в *Builder 6.0*). Имя формы нашего приложения – *Obuchn_Klassu*, на форме имеется единственная командная кнопка с именем *Button1*. Код с описанием пользовательских классов нашего приложения следует расположить в упомянутом заголовочном файле *между* следующими двумя строками:

```
extern PACKAGE TObuchn_Klassu *Obuchn_Klassu;  
#endif
```

На указанных строках заголовочного файла поместим описание пользовательских классов и глобальную константу.

```
const Cen_Bal= 25;//Стоимость балла в гривнях (рублях, евро и др.)  
//Описание класса для человека, который получает только оклад  
class TChel//Общий предок для всех классов группы  
{  
    protected://Защищённый раздел  
        String FIO;//Фамилия, Имя, Отчество  
        float Sym_Vupl;//Сумма выплат  
    public://Открытый раздел  
        void Inic(String F_I_O, float Okl);//Инициализация полей
```

```

    void Vuvod(void); //Вывод строки платёжной ведомости,
    //наследуется всеми потомками
}; //TChel

class TStud: public TChel //Открытое наследование
{
    private: //Закрытый раздел
        float Sr_Bal; //Средний балл сессии
    protected:
        void Ras_Vupl(void); //Расчёт выплат
    public:
        void Inic(String F_I_O, float Sredn_B);
}; //TStud

class TSotr: public TChel
{
    private:
        float Razm_Prem;
    protected:
        void Ras_Vupl(void); //У предка этот метод отсутствует
    public:
        void Inic(String F_I_O, float Okl, float Razm_Pr);
}; //TSotr

class TPrep: public TSotr //TChel наследуется неявно через
                           //TSotr, можно TChel указать явно
{
    private:
        int Chasu; //Величина месячной почасовой нагрузки
        //Стоимость часа занятий (для ассистентов, доцентов и
        //профессоров в разных вузах может быть различной)
        float Stoim_Ch; //Стоимость часа
    protected:
        void Ras_Vupl(void); //Переопределение метода
    public: //Переопределение метода
        void Inic(String F_I_O, float Okl, float Razm_Pr,
                    int Cha, float St_Chasa);
}; //TPrep

```

Текст программы, размещаемый в файле реализации

```

void TChel::Vuvod(void) //Общий метод
{
    ShowMessage(FIO + FloatToStrF(Sym_Vupl, ffFixed, 8, 2));
} //TChel::Vuvod

void TStud::Ras_Vupl(void)
{
    Sym_Vupl = Cen_Bal * Sr_Bal; //Наследуется всеми потомками
} //TStud::Ras_Vupl

void TSotr::Ras_Vupl(void)
{
    Sym_Vupl = Sym_Vupl + Razm_Prem;
} //TSotr::Ras_Vupl

```

```

void TPrep::Ras_Vupl(void)
{
    Sym_Vupl= Sym_Vupl + Chasu*Stoim_Ch;
} //TPrep::Ras_Vupl

void TChel::Inic(String F_I_O, float Okl)
{
    FIO= F_I_O;
    Sym_Vupl= Okl;
} //TChel::Inic

void TStud::Inic(String F_I_O, float Sredn_B)
{
    FIO= F_I_O;
    Sr_Bal= Sredn_B;
    TStud::Ras_Vupl();
} //TStud::Inic

void TSotr::Inic(String F_I_O, float Okl, float Razm_Pr)
{
    TChel::Inic(F_I_O, Okl); //Доступ к методу предка
    Razm_Prem= Razm_Pr;
    TSotr::Ras_Vupl();
} //TSotr::Inic

void TPrep::Inic(String F_I_O, float Okl, float Razm_Pr,
                  int Cha, float St_Chasa)
{
    TSotr::Inic(F_I_O, Okl, Razm_Pr); //Доступ к методу предка
    Chasu= Cha;
    Stoim_Ch= St_Chasa;
    TPrep::Ras_Vupl();
} //TPrep::Inic

TChel Chel; //Объявление объектов Chel, Stud, Sotr и Prep
TStud Stud;
TSotr Sotr;
TPrep Prep;

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Chel.Inic("Иванов Иван Иванович ", 813.6);
    Stud.Inic("Петров Пётр Петрович ", 4.7);
    Sotr.Inic("Сидоров Сидор Сидорович ", 897.35, 1911);
    Prep.Inic("Яковлев Яков Яковлевич ", 1911, 1106.66, 78, 8.17);
} //FormCreate

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Chel.Vuvod(); //Иванов Иван Иванович 813.6
    Stud.Vuvod(); //Петров Пётр Петрович 117.50
    Sotr.Vuvod(); //Сидоров Сидор Сидорович 2808.35
    Prep.Vuvod(); //Яковлев Яков Яковлевич 3654.92
} //Button1Click

```

В комментариях в функции *Button1Click* показаны выводимые на экран строки ведомости.

Обращаем внимание на то, что в методах *Inic* имена формальных параметров *обязательно* должны *отличаться* от имён тех полей, которые эти формальные параметры инициализируют. В противном случае (когда они одноимённы) программа работает неправильно: выводит ошибочные результаты. Поэтому внимательно следите за тем, чтобы параметры методов классов не были одноимёнными с соответствующими полями классов.

Поскольку в данном варианте программы используются обычные переменные-объекты, то доступ к их полям и методам осуществляется при помощи операции *точка*. В случае указателей на динамические объекты для достижения прежних действий применяется операция *стрелка*.

Порождаются указатели на динамические объекты при помощи операции *new*:

```
TChel *Chel= new TChel;
TStud *Stud= new TStud;
TSotr *Sotr= new TSotr;
TPrep *Prep= new TPrep;
```

Вышеприведенные четыре строки размещаются в файле реализации (не обязательно после описания методов классов *TChel*, *TStud*, *TSotr* и *TPrep*).

В этом случае функции *FormCreate* и *Button1Click* выглядят так:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Chel->Inic("Иванов Иван Иванович ", 813.6);
    Stud->Inic("Петров Пётр Петрович ", 4.7);
    Sotr->Inic("Сидоров Сидор Сидорович ", 897.35, 1911);
    Prep->Inic("Яковлев Яков Яковлевич ", 1911, 1106.66, 78, 8.17);
} //FormCreate

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Chel->Vuvod(); //Иванов Иван Иванович 813.6
    Stud->Vuvod(); //Петров Пётр Петрович 117.50
    Sotr->Vuvod(); //Сидоров Сидор Сидорович 2808.35
    Prep->Vuvod(); //Яковлев Яков Яковлевич 3654.92
} //Button1Click
```

К полям и методам указателей на динамические объекты можно ещё обращаться с использованием операции разыменования и операции *точка*: *(*Prep).Vuvod()*;

В обоих вариантах программы (применение объектов и указателей на тип объектов) в описании классов используются *только* прототипы методов, описание методов приводится в файле реализации. В качестве базового класса использован класс человека, у которого сумма выплат запоминается в поле *Sym_Vupl*. Инициализация этого поля в объекте *Chel* осуществляется величиной оклада *Ok* при помощи метода *Inic*. Лица такой категории (типа *TChel*) *не были* предусмотрены условием задачи, однако в ООП *рекомендуется* в качестве базового предлагать такой класс, который *может* использоваться в качестве предка для *всех* остальных классов группы. На основе такого базового класса создаётся *абстрактный* класс, процедура его разработки и использования подробно разъясняется в п. 14.4.3.

Класс *TPrep* наследует класс *TChel* неявно – через класс *TSotr*. Не является ошибкой перечислить в определении класса *TPrep* оба базовые классы *TChel* и *TSotr*, осуществить *множественное наследование* классов. Если производный класс имеет всего лишь *одного* явного родителя, то такое наследование называют *простым наследованием*.

Метод *Vuvod()* открытого раздела класса человека *TChel* наследуется *всеми* потомками, поскольку для всех дочерних классов он пригоден для вывода двух защищённых полей *FIO* и *Sym_Vupl*, требуемых в платёжной ведомости. Однако *расчёт* выплат для *различных* категорий лиц осуществляется по *разным* формулам, поэтому в классах *TStud*, *TSotr* и *TPrep* метод *Ras_Vupl* реализуется по-разному. В классе *TChel* нет метода для расчёта выплат, поэтому у его наследников – в классах *TStud* и *TSotr* метод *Ras_Vupl* *добавляется* к методам родителя, а в классе *TPrep*, который *явно* наследует класс *TSotr*, этот метод *переопределяется* – имеет *тот же* заголовок, но *другое* тело.

Для инициализации полей экземпляров классов применяется метод *Inic*. Этот метод во всех дочерних классах переопределяется (и имеет разное число параметров). В дочерних классах можно применять как наследуемые, так и переопределяемые методы.

Обращаем внимание на то, что в переопределённых методах инициализации имеется возможность применять одноимённые методы предков. Эта возможность реализуется при помощи операции расширения области действия «::». Эту же операцию ещё называют уточнением области действия или операцией указания области видимости. Вот так, например, вызывается метод *Inic*, разработанный для класса *TSotr*, в теле метода инициализации *Inic* для класса *TPrep*:

```
TSotr::Inic(F_I_O, Okl, Razm_Pr);
```

Такое использование *одноимённых* методов родительских классов считается *хорошим стилем* программирования, особенно для инициализирующих методов и конструкторов (разговор о них впереди). Это способствует структурированию программ, сокращает их код и в целом повышает быстродействие.

В разработанном приложении все наследуемые классы открыты – используются с идентификатором доступа *public*. Однако производный класс может иметь защищённых (*protected*) и закрытых (*private*) родителей (содержать в себе защищённые и закрытые базовые классы). Следует признать, что защищённое и закрытое наследования встречаются редко. Вместе с тем познакомимся с возможностями *всех* типов наследования.

Открытое наследование родительского класса приводит к тому, что в производном классе открытые элементы (поля и методы) родителя являются также открытыми, а защищённые элементы класса-предка остаются защищёнными элементами и в дочернем классе. В производном классе при таком наследовании недоступны лишь закрытые элементы предка.

Защищённое наследование обеспечивает идентификатор доступа *protected* к элементам производного класса, перешедшим в него из базового класса со спецификаторами доступа *public* и *protected*.


```

class TSotr: public TChel
{
    protected:
        void Ras_Vupl(void); //У предка этот метод отсутствует
    public: //Переопределённый метод
        void Inic(String F_I_O, float Okl, float Razm_Pr,
                    int Cha, float St_Chasa);
}; //TSotr

class TPrep: public TSotr
{
    protected:
        void Ras_Vupl(void); //Переопределение метода
    public: //Переопределение метода
        void Inic(String F_I_O, float Okl, float Razm_Pr,
                    int Cha, float St_Chasa);
}; //TPrep

```

В файле реализации переопределяемые методы *Inic* для классов *TSotr* и *TPrep* заменяются такими вариантами.

```

void TSotr::Inic(String F_I_O, float Okl, float Razm_Pr,
                  int Cha, float St_Chasa)
{
    TChel::Inic(F_I_O, Okl, 0, 0, 0); //Доступ к методу предка
    Razm_Prem= Razm_Pr;
    TSotr::Ras_Vupl();
} //TSotr::Inic

void TPrep::Inic(String F_I_O, float Okl, float Razm_Pr,
                  int Cha, float St_Chasa)
{
    TSotr::Inic(F_I_O, Okl, Razm_Pr, 0, 0 ); //Метод предка
    Chasu= Cha;
    Stoim_Ch= St_Chasa;
    TPrep::Ras_Vupl();
} //TPrep::Inic

```

При инициализации объектов типов *TSotr* и *TPrep* в методах *Inic* для неиспользуемых в них полей (например, *Cha* и *St_Chasa*) формальные переменные заменяются нулевыми значениями (в принципе, можно использовать *любые* допустимые числовые значения).

В файле реализации введём глобальный указатель *Mas_Lydi* на массив (из четырёх элементов) типа *TChel*.

```
TChel *Mas_Lydi[4];
```

Тело функции *FormCreate* теперь станет таким:

```

Mas_Lydi[0]= new TChel;
Mas_Lydi[1]= new TStud;
Mas_Lydi[2]= new TSotr;
Mas_Lydi[3]= new TPrep;

Mas_Lydi[0]->Inic("Иванов Иван Иванович ", 813.6, 0, 0, 0);
Mas_Lydi[1]->Inic("Петров Пётр Петрович ", 4.7, 0, 0, 0);
Mas_Lydi[2]->Inic("Сидоров Сидор Сидорович ", 897.35, 1911, 0, 0);

```

```
Mas_Lydi[3]->Inic("Яковлев Яков Яковлевич ", 1911, 1106.66, 78, 8.17);
```

В этой функции в *каждый* элемент указателя *Mas_Lydi* на массив записываются адреса экземпляров класса соответственно для *TChel*, *TStud*, *TSotr* и *TPrep*. Поскольку только в экземпляре класса *TPrep* все поля инициализируются конкретными значениями, поэтому в других объектах (классов *TChel*, *TStud* и *TSotr*) значения неиспользованных числовых полей заменяются нулями (как отмечалось выше, можно использовать также *произвольные* допустимые числовые значения).

Прежнюю функцию *Button1Click* заменим следующим вариантом:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    for (int ij= 0; ij< 4; ++ij)
        Mas_Lydi[ij]->Vuvod();
} // Button1Click
```

Эта функция *последовательно* выводит на экран строки платёжной ведомости, однако сумма выплат окажется верной *только в первой строке*. Для объяснения причины неправильной работы этого варианта приложения познакомимся вначале с правилами совместимости экземпляров классов.

Вопросы для самоконтроля

- ☐ Что нужно предпринять для того, чтобы программу со структурами преобразовать в приложение с классами?
- ☐ Зачем используются спецификаторы доступа *public*, *protected*, *private* и *published*?
- ☐ Могут ли классы быть вложенными друг в друга?
- ☐ Что называют переопределением методов в классах и структурах?
- ☐ Как обратиться к методу предка, если он переопределён у потомка?
- ☐ Как осуществляется множественное наследование в потомках классов?
- ☐ Зачем применяется инкапсуляция? Приведите примеры её использования.
- ☐ Поля находятся в классах или в экземплярах классов? Где располагаются методы?
- ☐ Почему описания классов и реализации их методов рекомендуется приводить в разных файлах? Какие это файлы?
- ☐ Спецификаторы доступа в классе (*public*, *protected* и др.) относятся к элементам класса, расположенным за ними. Где находится нижняя граница действия спецификаторов?
- ☐ Почему инициализирующие методы дочерних классов обычно используют инициализирующие методы предков, а не переопределяют весь процесс инициализации заново?
- ☐ Наследуется ли метод предка, если он не переопределён в классе-наследнике? Можно ли в классе-потомке одновременно применять переопреде-

лённые методы и методы предков с такими же именами?

- ☐ В каких случаях доступ к полям и методам осуществляется через операцию стрелка «->», а когда при помощи операция точка «.»?
- ☐ Что называется простым наследованием?
- ☐ Какие права доступа имеются у производного класса для тех его элементов, которые унаследованы при *открытом* наследовании родительских классов?
- ☐ Какие права доступа имеются у унаследованных элементов при *защищённом* (*закрытом*) наследовании?
- ☐ В каком порядке описываются поля и методы классов? Вначале описываются поля, а затем методы, или наоборот?

14.4. Полиморфизм

Как отмечалось, полиморфизм является одной из основ ООП-технологии. Для его детального изучения познакомимся с рядом понятий и правил.

14.4.1 Совместимость объектных типов

Правила совместимости типов объектов легко запомнить и понять их логичность после знакомства с *принципом совмещения*: поля данных источника должны *полностью* заполнять поля данных приёмника. В этом случае в операторах присваивания из источника в приёмник копируются *только поля*, являющиеся *общими* для обоих классов (или типов). Так, например, для первого варианта приложения (прежнего устройства классов) в операторе присваивания

```
Chel = Prep;
```

только поля *FIO* и *Sym_Vupl* из объекта *Prep* копируются в объект *Chel*, поскольку *только они* являются общими для *Chel* и *Prep*.

Указатель на класс потомка разрешается присваивать указателю на класс родителя. Для иллюстрации этого правила введём указатели на ранее объявленные классы:

```
TChel * Yk_Chel = new TChel;  
TStud * Yk_Stud = new TStud;  
TSotr * Yk_Sotr = new TSotr;  
TPrep * Yk_Prep = new TPrep;
```

Теперь являются допустимыми следующие операторы присваивания:

```
Yk_Sotr = Yk_Prep;  
Yk_Chel = Yk_Sotr;  
Yk_Chel = Yk_Prep;
```

Формальный параметр функции заданного объектного типа может принимать в качестве фактического параметра объекты *своего же типа* или объекты *всех дочерних типов*:

```
void Primer_Prototipa_Fynkcii(TSotr Sotrydnichek);
```

В этой функции допустимыми типами фактического параметра (который заме-

няет параметр-значение *Sotrydnichek*) могут быть *TSotr*, *TPrep*, но не *TChel*. В рассматриваемом случае это *копия* фактического параметра, включающая *только* поля, имеющиеся в типе формального параметра. Таким образом, копия фактического параметра *преобразовывается* к типу формального параметра, значение же самого фактического параметра остаётся *неизменным*.

Если формальный параметр функции является указателем на какой-то конкретный тип (на класс), то допустимыми фактическими параметрами являются указатели на этот же тип или на любой его производный тип. В качестве примера рассмотрим следующий прототип функции:

```
void Prim_Prototip_Fynk (TSotr * Yk_Sotr);
```

В этой функции допустимы следующие типы фактического параметра *TSotr**, *TPrep**. Переменные типа *TChel** (например, *Yk_Chel*) не могут применяться в качестве фактических параметров функции *Prim_Prototip_Fynk*.

Сформулируем теперь в кратком виде правила совместимости объектных типов. Совместимость выполняется:

- ☐ между экземплярами класса;
- ☐ между указателями на экземпляры класса;
- ☐ между согласованными формальными и фактическими параметрами функций.

Во всех перечисленных случаях совместимость типов расширяется только от потомка к родителю: дочерние типы разрешается использовать взамен родительских, но не наоборот.

Вернёмся теперь ко второму варианту приложения (см. раздел 14.3). Присваивание разным элементам массива *Mas_Lydi* типа **TChel* адресов объектов типов **TChel*, **TStud*, **TSotr* и **TPrep* вполне допустимо, поскольку это согласуется с рассмотренными выше правилами совместимости. Однако согласно этим же правилам в порождённые объекты, адреса которых записаны в элементы массива, будут передаваться *только* те поля и методы, которые *имеются* у базового типа *TChel*. Именно по этой причине в платёжной ведомости, выведенной модернизированной программой, сумма выплат для *всех лиц* будет определяться также как и для типа *TChel*. Поэтому, рассмотренный ранее механизм наследования, поставленную задачу в данном варианте приложения решить *не может*.

14.4.2. Полиморфизм, виртуальные методы, раннее и позднее связывание

Слово *полиморфизм* происходит от греческих слов *poly* (много) и *morphos* (форма), означает множественность форм. При использовании классов полиморфизм заключается в *гибкости* возможностей функций и классов, проявляется в разных ситуациях.

- ☐ Согласно правилу совместимости объектных типов формальному параметру функции может соответствовать фактический параметр не только того же типа, что и формальный, но и типа, производного от него. Такой параметр называется *полиморфным*.

- Примером полиморфизма является переопределение методов в дочерних классах, что позволяет производить отличающиеся операции (действия) при вызове функции с одним и тем же именем.

Однако в полной мере полиморфизм объектов и методов реализуется при помощи использования *виртуальных методов*. Поэтому достаточно часто в литературе под полиморфизмом понимают *только* те возможности классов, которые появляются в случае применения виртуальных методов. Виртуализация методов – это мощный механизм, обеспечивающий гибкое использование кодов переопределённых методов в дочернем классе и во всех его предках. Последнее предложение, видимо, воспринимается читателями пока лишь, как бессмысленный набор слов.

Для прояснения сущности виртуальных методов рассмотрим вначале механизм *раннего* или *статического связывания*. То, о чём говорится ниже справедливо как для переопределённых, так и непереопределённых методов. Однако механизм виртуализации целесообразен *только* для переопределённых методов.

При порождении экземпляра класса с обычными (не виртуальными) переопределёнными методами *на этапе компиляции* программы устанавливаются *связи* объекта с методами класса (напоминаем, что методы хранятся не в объектах, а в их классе). Поэтому в ходе выполнения программы при обработке определённого поля объекта вызывается из класса именно тот метод, с которым установлена связь ещё в период трансляции программы. Процесс, с помощью которого вызовы методов связываются с объектом во время компиляции программы, называется *ранним связыванием*.

В механизме *позднего* или *отложенного*, или *динамического связывания* компилятор откладывает эту связь до момента выполнения программы (поэтому оно и называется *поздним*). Такой механизм применим *только* для *динамических объектов*. Прежде, чем познакомиться с его сущностью, напомним, что указателю на базовый класс, согласно правилам совмещения типов, можно присвоить значение адреса объекта этого же класса или *любого* его *производного* класса и что вызов методов объекта происходит в соответствии с *типом указателя*, а не *фактического типа объекта*, на который он ссылается.

Поэтому во втором варианте учебного приложения, не смотря на то, что при инициализации в ячейки массива *Mas_Lydi* последовательно записывались адреса экземпляров базового класса и всех его потомков, для обработки поля вызывался только метод базового класса *TChel*. Как уже отмечалось, это обусловлено тем обстоятельством, что в ходе трансляции программы компилятор (согласно объявлению массива *Mas_Lydi*) связал каждую ячейку массива с методом *базового* класса. Поэтому, даже явное преобразование типов при инициализации ячеек массива *Mas_Lydi* не меняет эту связь:

```
Mas_Lydi[0]= new TChel;  
(TStud*)Mas_Lydi[1]= new TStud;  
(TSotr*)Mas_Lydi[2]= new TSotr;  
(TPrep*)Mas_Lydi[3]= new TPrep;
```

Вместе с тем, если впереди переопределяемого метода в описании классов поста-

вить служебное слово *virtual*, то программа будет выводить правильные суммы выплат для лиц *всех* категорий. При этом вполне достаточно упомянутое ключевое слово поставить *только* перед *первым* переопределяемым методом – все последующие одноимённые методы также будут виртуальными. Таким образом, если в описании классов при первом упоминании прототипов методов *Ras_Vupl* и *Inic* впереди этих прототипов указать спецификатор *virtual*, то все последующие переопределения названных методов будут считаться виртуальными. Однако согласно правилам хорошего стиля программирования рекомендуется впереди *всех* последующих переопределяемых методов также ставить слово *virtual*. Такой повтор слова *напомнит* программисту (при последующей модернизации приложения) истинный вид конкретного метода, к этому особенно полезно прибегать, если дочерний и родительский классы определяются *в разных* программных модулях. Переопределяемые методы можно сделать виртуальными только в том случае, когда *идентичны* списки их параметров в заголовках и прототипах (число параметров и очерёдность их типов совпадают).

Как уже отмечалось, возможности виртуализации переопределяемых методов обеспечивает механизм позднего связывания. Каждый класс, содержащий виртуальные методы, имеет таблицу виртуальных методов (ТВМ), хранящуюся в сегменте данных. В ТВМ содержится значение величины объёма, занимаемым классом, и указатели на начала кодов реализации *всех* виртуальных методов класса. Адреса виртуальных методов содержатся в таблице в порядке их описания в классе.

При инициализации объекта с виртуальными методами устанавливается связь между экземпляром класса и ТВМ класса. Важно отметить, что имеется *только одна* ТВМ для *каждого* класса. Отдельные экземпляры класса (объекты) содержат только адрес ТВМ. Указанный адрес записывается в объекты ещё на этапе трансляции программы – в ходе её компиляции. Это оказывается возможным благодаря тому, что в *каждом* объекте имеется дополнительное *скрытое поле ссылки*, предназначенное для записи упомянутого адреса. На этапе компиляции программы ссылки на виртуальные методы заменяются адресом ТВМ, который берётся из скрытого поля объекта, а на этапе выполнения программы, в момент обращения к методу, его адрес выбирается из таблицы. Таким образом, вызовы виртуальных методов, в отличие от вызовов обычных методов, выполняется *через дополнительный этап* получения адреса метода из ТВМ, что несколько замедляет выполнение программы.

Итак, ссылка на виртуальный метод производится *на этапе выполнения* программы, то есть по факту вызова. Перевод слова *virtual* – фактический, действительный, поэтому метод с таким спецификатором можно также называть *фактическим*.

Виртуальный механизм применим *только* для динамических объектов. Указатель на динамический объект, в типе которого содержится виртуальный метод (или методы), называется *полиморфным*. В данном случае полиморфизм состоит в том, что с помощью обращения к выбранному методу выполняются *различные*

действия в зависимости от типа, на который ссылается указатель в каждый момент времени. Утверждения, упомянутые выше, видимо, не до конца ясны или же совершенно непонятны. Их рекомендуем повторно прочесть после подробного рассмотрения нижеприведенных примеров.

14.4.3. Чисто виртуальные методы, абстрактные классы и полиморфные функции

В учебном приложении не случайно описан класс *TChel*, не *требуемый* по условию задачи. Объявление таких классов позволяет описать *чисто виртуальные методы*. Добавим в класс *TChel* чисто виртуальный метод:

```
virtual void Ras_Vupl(void) = 0;
```

Этот метод *не способен* выполнять какое-либо действие, он *обязательно* должен переопределяться в производных классах. При этом в классах-потомках его разрешается переопределить ещё раз, как чисто виртуальный метод.

Класс, содержащий хотя бы один чисто виртуальный метод, называется *абстрактным*. Абстрактные классы предназначены для описания *общих* понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для дочерних классов, поскольку объекты абстрактного класса создавать *запрещается*, вызов чисто виртуального метода приводит к ошибке выполнения программы. Применение абстрактных классов связано со следующими ограничениями и правилами:

- ❑ абстрактный класс *нельзя* использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;
- ❑ допускается объявлять указатели и ссылки на чисто абстрактный класс, если при инициализации не требуется создавать объекты этого класса;
- ❑ если в классе, производном от абстрактного класса, не определяются *все* чисто виртуальные функции, то он также является абстрактным.

Согласно этим правилам можно разработать функцию, параметром которой является указатель на абстрактный класс. В ходе выполнения программы в этот параметр можно передавать указатель на объект *любого производного класса*. Таким образом создаются *полиморфные функции*, способные в пределах *одной иерархии* производного класса работать с объектами *любого* родительского типа.

14.4.4. Применение виртуальных и чисто виртуальных методов, правила их описания и использования

Ниже приведена та часть кода последнего варианта учебного приложения, которая претерпела изменения, выполненные с целью иллюстрации применения чисто виртуальной функции и абстрактного класса. Класс *TChel* сделаем абстрактным. При этом описание классов *TStud*, *TSotr*, *TPrep*, метода *Vuvod* и виртуальных методов *Ras_Vupl* и *Inic* останутся прежними.

```

class TChel
{
    protected:
        String FIO;
        float Sym_Vupl,
              Sr_Bal,
              Razm_Prem,
              Stoim_Ch;
        int Chasu;
    public:
        virtual void Inic(String F_I_O, float Okl, float Razm_Pr, int Cha,
                          float St_Chasa);

        void Vuvod(void);
        virtual void Ras_Vupl(void) = 0; // Чисто виртуальный метод
}; // TChel

```

/*Описания классов TStud, TSotr, TPrep, метода Vuvod и виртуальных методов Ras_Vupl и Inic остались прежними поэтому здесь не приводятся */

Тело функции *FormCreate*:

```

Mas_Lydi[1] = new TStud;
Mas_Lydi[2] = new TSotr;
Mas_Lydi[3] = new TPrep;

Mas_Lydi[1]->Inic("Петров Пётр Петрович ", 4.7, 0, 0, 0);
Mas_Lydi[2]->Inic("Сидоров Сидор Сидорович ", 897.35, 1911, 0, 0);
Mas_Lydi[3]->Inic("Яковлев Яков Яковлевич ", 1911.0, 1106.66, 78, 8.17);

```

Тело функции *Button1Click*:

```

for (int ij = 1; ij < 4; ++ij)
    Mas_Lydi[ij]->Vuvod();

```

От предшествующего варианта приложения функция *FormCreate* отличается лишь тем, что в ней *не создаётся* (поскольку это запрещено) и не инициализируется объект для чисто виртуального класса *TChel*. Именно по этой причине цикл *for* в функции *Button1Click* начинается не с нуля, а с единицы: вызов чисто виртуального метода приводит к ошибке на стадии выполнения программы.

Перечислим теперь правила описания и использования виртуальных методов:

- ❑ Если в базовом классе метод определяется как виртуальный, то в производном классе этот метод *автоматически* становится виртуальным только в случае, когда он определён с тем же именем и *набором параметров*. В производных классах *разрешается* объявлять методы, имена которых *совпадают* с именем виртуального метода, но имеют *другой набор* параметров. В этом случае такие методы являются обычными (не виртуальными методами).
- ❑ Виртуальные методы *наследуются*, поэтому переопределять их в производном классе требуется *только* при необходимости задать отличающиеся действия. Права доступа при переопределении метода изменять *нельзя*.
- ❑ Если виртуальный метод переопределяется в производном классе, то элементы этого класса могут получать доступ к любому методу базового клас-

са с помощью операции доступа к области видимости (операции расширения области видимости).

- ❑ Виртуальный метод не может объявляться с модификатором *static*, но может быть объявлен, как *дружественный* (эти понятия рассматриваются в следующих разделах).
- ❑ Если в классе вводится описание виртуального метода, то в файле реализации *необходимо* привести его код, *либо* в самом классе определить его как чисто виртуальный метод.

14.4.5. Преимущества и недостатки виртуальных методов

Не существует чётких правил, согласно которым метод следует объявлять виртуальным. Можно только рекомендовать делать метод виртуальным, когда имеется вероятность того, что он будет переопределён в новых производных классах при последующих модернизациях приложения. Методы, которые во всей иерархии остаются неизменными или методы, которые не применяются в производных классах, объявлять виртуальными не имеет смысла. Вместе с тем, на начальном этапе разработки приложения, не всегда имеется возможность предсказать в деталях пути расширения базовых классов, а объявление метода виртуальным обеспечивает *гибкость* дальнейшего расширения возможностей приложения.

Однако у класса с виртуальными методами создаётся ТВМ, и *любой* экземпляр класса связывается с этой таблицей. Поэтому *каждый* вызов виртуального метода проходит через ТВМ, тогда как обычные методы вызываются *непосредственно*, что несколько повышает быстродействие приложения.

14.5. Клиенты и дружественные функции класса

К открытым методам и полям класса имеют доступ не только методы класса, но и внешние функции. Функции, работающие с открытыми элементами класса, называются *клиентами класса*.

Внешние функции, которые имеют доступ к *закрытым* и *защищённым* (скрытым) полям класса, называются *дружественными функциями* или *друзьями класса*. Для доступа к скрытым полям класса внешняя функция объявляется в классе, как друг класса при помощи спецификатора *friend* (друг).

Если в нашем учебном приложении наследуемый метод *Vuvod()* заменить дружественной функцией с одноимённым именем, то в теле класса требуется включить прототип:

```
friend void Vuvod(TChel *Yk); //Дружественная функция
```

В файле реализации (или в каком-то другом модуле программы) необходимо

описать реализацию этой функции, например, так:

```
void Vuvod (TChel *Yk)
{
    ShowMessage(Yk->FIO + FloatToStrF(Yk->Sym_Vupl, ffFixed, 8, 2));
} //Vuvod
```

Теперь скорректируем и тело функции *Button1 Click*:

```
for (int ij= 1; ij< 4; ++ij)
    Vuvod(Mas_Lydi[ij]);
```

После упомянутых изменений приложение по-прежнему выводит строки платёжной ведомости, однако такой вывод осуществляется при помощи дружественной функции.

Таким образом, дружественные функции, в замен методов, используются для обработки скрытых полей класса. Перечислим правила описания и особенности дружественных функций:

- ❑ Дружественная функция объявляется с ключевым словом *friend* внутри того класса, к скрытым полям которого ей требуется разрешить доступ. Указатель *this* дружественной функции не передаётся, поэтому обязательным её параметром является *объект* или *ссылка* на объект заданного класса.
- ❑ Обычная функция или метод *другого* класса (определённых *ранее* класса с дружественной функцией) могут выступать в роли дружественной функции. Местоположение объявления дружественной функции в классе *не имеет значения*, поскольку на неё не распространяется действие спецификаторов доступа.
- ❑ *Несколько* классов могут иметь общую дружественную функцию.

Дружественные функции не рекомендуется использовать, поскольку они нарушают принцип инкапсуляции и поэтому затрудняют отладку и модернизацию приложений.

14.6. Статические поля и статические константы класса

В начале этого урока говорилось о том, что в отличие от методов, поля данных хранятся не в классе, а в экземплярах класса – в объектах. Однако это несколько не так. Дело в том, что программист *имеет возможность* выбранное поле (или поля) данных хранить *только* в классе, а в объект передавать лишь ссылку (ссылки) на него. В этом случае такое поле называется *статическим*, оно является *глобальным* для всех экземпляров класса, поэтому каждый объект упомянутого класса имеет свободный доступ к названному полю (способен его читать и изменять). Необходимость в глобальных переменных класса нередко возникает в ходе разработки программ, поэтому гибкий язык C++ предоставляет механизм реализации таких переменных.

В качестве примера в классе *TChel* расположим открытое поле *Ch_Ob*, которое будет служить счётчиком вызываемых в программе объектов. Сделаем так, чтобы

после *каждого* вызова объекта значение этого поля увеличивалось на единицу. Для реализации этой задачи в *открытом* разделе класса *TChel* объявим статическое поле. Описание статического поля начинается со служебного слова *static*:

```
static int Ch_Ob; //Открытое статическое поле
```

Статическое поле, предназначенное для тех же целей, что и поле *Ch_Ob*, расположим и в закрытом или защищённом разделе класса:

```
static int Skrut_Pole; //Скрытое статическое поле
```

Чтение и изменение значения *Skrut_Pole* выполняют специальные *статические* функции *Chten()* и *Inkr()*, прототипы которых располагаются в *открытом* разделе класса и предваряются ключевым словом *static*:

```
static int Chten(void);
```

```
static int Inkr(void);
```

В файле реализации приводятся описание этих функций:

```
int TChel::Chten(void)
{
    return Skrut_Pole;
} //TChel::Chten
```

```
int TChel::Inkr(void)
{
    return Skrut_Pole++;
} //TChel::Inkr
```

Статические функции могут работать *только* со статическими данными. Вне объявления класса (в заголовочном файле или в файле реализации) *обязательно* должны задаваться *начальные* значения статических полей. Стартовые значения таких статических переменных *запрещается* объявлять внутри функций. При этом *нет различий* в инициализации открытых, закрытых и защищённых статических полей:

```
int TChel::Ch_Ob= 0; //Очистка открытого статического поля
```

```
int TChel::Skrut_Pole= 0; //Очистка скрытого статического поля
```

На этапе компиляции программы выдаётся сообщение о неразрешённой внешней ссылке до тех пор пока *все* статические поля не получают начальные значения. Статическим полям-переменным разрешается задавать стартовые значения *только один раз*.

Пусть в файле реализации порождён объект:

```
TPrep Prepod;
```

Теперь доступ к открытому статическому полю можно осуществить двумя способами – через объект и посредством имени класса с помощью бинарной операции уточнения области действия:

```
ShowMessage(Prepod.Ch_Ob); //Первый способ
```

```
ShowMessage(TChel::Ch_Ob); //Второй способ
```

Вывод значений *закрытых* и *защищённых* статических полей осуществляется с использованием упомянутой выше статической функции чтения *Chten*:

```
ShowMessage(Prepod.Chten());
```

```
ShowMessage(TChel::Chten());
```

Вот такой вид может иметь модернизированная функция *Button1Click* учебной программы (с виртуальными функциями и абстрактными классами), иллюстрирующей применение статических полей:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    for (int ij= 1; ij< 4; ++ij)
    {
        Massiv_Lydi[ij]->Vuvod();
        TChel::Ch_Ob++; //Открытый счётчик числа вызовов объектов
        TChel::Inkr(); //Закрытый счётчик числа вызовов объектов
    } //for_ij
    ShowMessage(TChel::Ch_Ob); //Выводится 3 (открытое поле)
    ShowMessage(TChel::ChTen()); //Выводится 3 (закрытое поле)
    //Варианты показа открытого и закрытого статических полей
    ShowMessage(Prep.Ch_Ob); //Выводится число 3
    ShowMessage(Prep.ChTen()); //Выводится число 3
} //Button1Click
```

Постоянные значения для *всех* экземпляров класса также неразумно тиражировать в каждом объекте. Поскольку это увеличивает объём кода и объём оперативной памяти. Значения *статических* именованных констант класса *могут* задаваться в *момент их объявления* в классе:

```
static const int Konst= 70904;
```

Можно также для их инициализации вне класса применить один из способов, показанный выше для инициализации статических полей-переменных, например такой:

```
const int TChel::Konst= 70904; //Менее удобный способ
```

При этом в самом классе при объявлении статической константы инициализация недопустима:

```
static const int Konst;
```

Последний способ объявления и инициализации статических констант очень уж неудобен: рекомендуем значения статическим константам присваивать сразу же в самом классе. Ниже показаны способы доступа к статическим константам (этот код можно расположить, например, в предшествующей функции *Button1Click*):

```
//Способы обращения к статическим константам
ShowMessage(TChel::Konst); //70904
ShowMessage(Prep.Konst); //70904
```

Вопросы для самоконтроля

- ☐ Назовите три разновидности полиморфизма.
- ☐ Что называют ранним связыванием.
- ☐ Объясните механизм позднего, отложенного или динамического связывания.
- ☐ Какой метод называется виртуальным?
- ☐ Нужно ли слово *virtual* ставить в дочерних классах переопределяемых виртуальных методов?

- ☐ Какие методы классов можно делать виртуальными?
- ☐ Зачем нужны чисто виртуальные методы?
- ☐ Можно ли порождать объекты абстрактных классов, допустим ли указатель на абстрактный класс?
- ☐ Как называется функция, параметром которой является указатель на абстрактный класс? Почему она имеет такое название? Может ли эта функция работать с объектами любого родительского класса?
- ☐ В каких случаях следует переопределять виртуальные методы?
- ☐ Какой метод вызывается быстрее виртуальный или обычный? Почему?
- ☐ Как и в каких случаях можно обработать поля класса внешними функциями?
- ☐ Чем дружественная функция отличается от клиента класса?
- ☐ Зачем применяются статические поля и статические константы класса?
- ☐ Назовите различия в инициализации открытых и скрытых статических полей.
- ☐ Зачем нужны статические функции?
- ☐ Как при помощи имени класса осуществить доступ к статической константе выбранного объекта?

14.7. Конструкторы класса

До настоящего времени пользовательские классы не имели конструкторов, поскольку их наличие в классе совершенно необязательно. *Конструктором класса* называется *открытый* метод, предназначенный для инициализации полей экземпляра класса. Конструктор вызывается автоматически (неявно) в момент порождения объекта или присваивания адреса указателю на тип объекта с использованием операции *new*.

Как и всякая функция, конструктор может иметь параметры, не иметь параметров и иметь параметры по умолчанию. Синтаксис для перечисленных случаев такой же, как и для обычных функций (он будет разъяснён ещё раз на последующих примерах). Ранее во всех разновидностях учебного приложения настоящего урока инициализация объектов осуществлялась не конструктором, а методом инициализации *Init*.

В отсутствие конструктора, клиенты класса (внешние функции, использующие открытые поля и методы класса) вынуждены *самостоятельно* присваивать полям класса их начальные значения. Однако это невозможно осуществить для закрытых и защищённых полей. В таком случае следует применять конструкторы. Дотошный читатель может не согласиться с последним утверждением. Послушайте, скажет он, во-первых, во всяком классе должно существовать *достаточное* количество методов для обработки *всех* его полей (закрытых, защищённых и открытых) для их чтения и записи в них допустимых значений,

во-вторых, прибегать к использованию клиентов класса и даже дружественных функций – это очень плохой стиль программирования, перечёркивающий все достоинства инкапсуляции. Зачем же применять конструкторы? В упомянутых случаях без конструкторов можно обойтись, однако их изучение (и применение) диктуется следующими причинами:

- ❑ конструкторы являются одной из основ ООП, поэтому о них следует знать всё;
- ❑ конструкторы применяются также для выделения памяти под динамические объекты, порождаемые *внутри* самого класса (поясняющие примеры приводятся ниже).

14.7.1. Конструкторы простых классов

Для первого знакомства с конструкторами рассмотрим *упрощённый* вариант прежнего приложения, в котором используется *только* один класс *TChel*. В коде приложения, расположенного ниже, применяется конструктор с параметрами *TChel(String F_I_O, float Okl)*. Найти метод, называемый конструктором, в описании любого класса очень просто: имя конструктора *совпадает* с именем класса и конструктор *не возвращает* никакого значения (даже тип *void*). Как уже отмечалось, конструктор всегда является *общедоступным* методом и предназначен для назначения полям объекта *стартовых* значений в момент порождения объекта или присваивания адреса указателю на тип объекта.

```
const Cen_Bal= 25;

class TChel
{
    protected:
        String FIO;
        float Sym_Vupl;
        void Ras_Vupl(float Okl);
    public:
        TChel(String F_I_O, float Okl); //Прототип конструктора
        void Vuvod(void);
}; //TChel::TChel

TChel::TChel(String F_I_O, float Okl) //Реализация конструктора
{
    FIO= F_I_O;
    TChel::Ras_Vupl(Ok1);
} //TChel::TChel

void TChel::Ras_Vupl(float Okl)
{
    Sym_Vupl= Okl;
} //TChel::Ras_Vupl

void TChel::Vuvod(void)
{
    ShowMessage(FIO + FloatToStrF(Sym_Vupl, ffFixed, 8, 2));
} //TChel::Vuvod
```



```
class TChel
{
    protected:
        String FIO;
```

```

float Sym_Vupl;
void Ras_Vupl(float Okl);
public:
    static const int Const= 909004;
    const int Konst;//Нестатическая константа
    TChel(void);//Прототип конструктора
    void Vuvod(void);
};//TChel::Tchel

TChel::TChel():Konst(110904)//Конструктор с инициализатором
{
    FIO= "Иванов Иван Иванович ";
    TChel::Ras_Vupl(813.6);
}//TChel::Tchel

TChel *Chel= new TChel();

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Chel->Vuvod();
    ShowMessage(TChel::Const);//90904
    ShowMessage(Chel->Konst);//110904
}//Button1Click

```

В реализации конструктора с инициализатором элементов после заголовка ставится двоеточие, далее приводится идентификатор константы, а за ним в круглых скобках указывается значение константы. При наличии в классе *нескольких* нестатических постоянных, все они перечисляются в заголовке реализации такого конструктора через запятую, значение каждой из них указывается в круглых скобках после соответствующего имени константы. Следует *особо подчеркнуть*, что в конструкторе с инициализатором можно инициализировать *только* нестатические константы, статические константы инициализируются указанными выше тремя способами.

Читатели могут задаться вопросом: насколько необходимы на практике столь изощрённые возможности задания констант? Программировать, конечно, вполне можно и без применения *всего* богатого арсенала инструментария *Builder*. Однако гибкие возможности среды программирования вы обязательно реализуете в *последующей* практике. При этом неизученная часть средств *Builder*, по-видимому, будет иметься *всегда*. Для уменьшения объёма этой непознанной части переходим к следующему пункту пособия.

14.7.2. Конструкторы производных классов

Если класс простой (не имеет предков) или наследует только абстрактные классы, то реализация его конструктора может осуществляться как в самом классе, так и вне класса. Однако конструкторы сложных классов (с базовыми классами) *не могут* описываться *вне* класса и имеют достаточно запутанный синтаксис. Для того чтобы в нём разобраться рассмотрим очень упрощённый пример.

Имеется базовый класс *A*, его наследует класс *B*, а класс *B* наследуется классом *C*. При этом все наследования являются открытыми. В коде программы, при-

```

class A
{
    public:
        int ia;
        A(int iaa);
}; //A

A::A(int iaa) //Реализация конструктора класса A
{
    ia = iaa;
}; //A::A

A Obj_A(3); //Инициализировали поле ia
class B: public A
{
    public:
        int ib;
        //Конструктор класса B, порождение и инициализация
        //родительского класса A
        B(int ibb):A(8){ib = ibb;}; //Новая инициализация поля ia
}; //B

B Obj_B(5); //Инициализировали поле ib
class C: public B
{
    public:
        int ic;
        //Конструктор класса C и новая инициализация поля ib
        C(int ibb, int icc):B(10){ib = ibb, ic = icc;};
}; //C
//Новая инициализация поля ib и первая
//инициализация поля ic
C Obj_C(11, 9);

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    ShowMessage(Obj_A.ia); //3
    ShowMessage(Obj_B.ia); //8
    ShowMessage(Obj_B.ib); //5

    ShowMessage(Obj_C.ia); //8 — прежнее значение поля
    ShowMessage(Obj_C.ib); //11 — новое значение поля
    ShowMessage(Obj_C.ic); //9 — новое значение поля
} //Button1Click

```

При порождении объекта *Obj_A* выполнена инициализация его единственного поля *ia*: *A Obj_A(3)*; Инициализация осуществляется указанием в круглых скобках после имени объекта допустимого значения поля. При порождении объекта *Obj_B* осуществляется инициализация его второго поля *ib*: *B Obj_B(5)*;

При этом значение поля *ia* осталось прежним (*ia*==3), тем же, что и в объекте *Obj_A*. Следует отметить, что, не смотря на то, что в объекте *Obj_B* имеется *два* поля, инициализировать при порождении этого объекта можно *только* поле *ib* (поле *ia* получило свое *новое* значение 8 при *объявлении* класса *B*). Поэтому вот такое порождение объекта *Obj_B* является ошибкой:

```
B Obj_B(5, 77);
```

Для описания конструктора дочернего класса после его заголовка (с параметрами или без них) следует поставить двоеточие, за которым после имени родительского класса (в нашем примере класса *A*) в круглых скобках привести допустимое значение поля (или полей) базового класса. Таким образом, как видно из вышеприведенного примера, для создания *любого* дочернего класса *необходимо вначале* в самом его конструкторе создать и инициализировать родительский класс. При этом полям родительского класса можно назначить как прежние, так и новые допустимые значения. В нашем примере поле *ia* при объявлении класса *B* получило новое значение, равное 8, а полю *ib* присвоено значение 5 при порождении объекта *Obj_B*. В конструкторе дочерних классов при порождении *каждого* дочернего класса предшествующий родительский класс *обязательно* необходимо порождать и инициализировать (или присваивать его полю или полям новые допустимые значения). Компилятор *не может* создать самостоятельно родительский класс, не смотря на то, что он указан как базовый наследуемый класс в объявлении класса.

В классе *C* иллюстрируется возможность присваивания *новых* значений выбранным полям (в примере только одно поле) родительского класса (или классов) *в момент создания* экземпляров этого класса. Эта возможность реализуется следующим образом – в дочернем классе объявляются как новое поле, так и поле, принадлежащее одному из родительских классов (в примере поле *ibb*):

```
C(int ibb, int icc):B(10){ib= ibb, ic= icc};
```

Теперь при порождении объекта класса *C* можно инициализировать не только его дополнительное поле, но и присвоить новое значение полю, принадлежащему родительскому классу *B*, которому назначается допустимое значение в конструкторе класса *C*. Это назначаемое в конструкторе значения поля (10) заменяется новым значением (11) в момент создания экземпляра класса *C*: *C Obj_C(11, 9)*; В нашем примере переопределено поле *ib*, но можно переопределить и поле *ia* или только поле *ia*: ограничений здесь никаких нет. Переопределять *все* поля объекта в момент его создания значительно удобнее: не требуется вносить изменения в код реализации конструктора, в программе наглядно видно какие же значения имеются у полей конкретного экземпляра класса. Но при желании, конечно же, можно применить и более запутанный способ инициализации объекта.

Внимательно просмотрите код приложения и последующие пояснения, осознайте правильность последовательно выводимых на экран значений полей объектов различных классов, осуществляемые функцией *Button1Click*. Эти значения указаны в виде комментариев к соответствующим строкам кода.

```
const Cen_Bal= 25;

class TChel
{
protected:
    String FIO;
    float Sym_Vupl;
public:
    void Vuvod(void);
    virtual void Ras_Vupl(void)= 0;
}; //Tchel

class TStud: public TChel
{
private:
    float Sr_Bal;
protected:
    virtual void Ras_Vupl(void);
public:
    TStud(String F_I_O, float Sredn_B);
}; //Tstud

class TSotr: public TChel
{
protected:
    float Razm_Prem;
    virtual void Ras_Vupl(void);
public:
    TSotr(String F_I_O, float Okl, float Razm_Pr);
}; //Tsotr

class TPrep: public TSotr
{
private:
    int Chasu;
    float Stoim_Ch;
protected:
    virtual void Ras_Vupl(void);
public:
    TPrep(String F_I_O, float Okl, float Razm_Pr, int Cha,
           float St_Chasa):TSotr("Федоренко Пётр Тихонович ",
                                   1911.0, 1106.66)
    {
        FIO= F_I_O;
        Sym Vupl= Okl;
    }
};
```

```

        Razm_Prem= Razm_Pr;
        Chasu= Cha;
        Stoim_Ch= St_Chasa;
        TPrep::Ras_Vupl();
    }; //TPrep
}; //Tprep

TStud::TStud(String F_I_O, float Sredn_B)
{
    FIO= F_I_O;
    Sr_Bal= Sredn_B;
    TStud::Ras_Vupl();
} // TStud::Tstud

TSotr::TSotr(String F_I_O, float Okl, float Razm_Pr)
{
    FIO= F_I_O;
    Sym_Vupl= Okl;
    Razm_Prem= Razm_Prem;
    TSotr::Ras_Vupl();
} // TSotr:: Tsotr

void TChel::Vuvod(void)
{
    ShowMessage(FIO + FloatToStrF(Sym_Vupl, ffFixed, 8, 2));
} //TChel::Vuvod

void TStud::Ras_Vupl(void)
{
    Sym_Vupl= Cen_Bal*Sr_Bal;
} //TStud::Ras_Vupl

void TSotr::Ras_Vupl(void)
{
    Sym_Vupl= Sym_Vupl + Razm_Prem;
} //TSotr::Ras_Vupl

void TPrep::Ras_Vupl(void)
{
    Sym_Vupl= Sym_Vupl + Razm_Prem + Chasu*Stoim_Ch;
} //TPrep::Ras_Vupl

TChel *Mas_Lydi[4];

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    for (int ij= 1; ij< 4; ++ij)
    {
        Mas_Lydi[ij]->Vuvod();
    } //for_ij
} //Button1Click

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Mas_Lydi[1]= new TStud("Петров Пётр Петрович ", 4.7);
    Mas_Lydi[2]= new TSotr("Сидоров Сидор Сидорович ", 897.35, 1911);

```

```
Mas_Lydi[3]= new TPrep("Федоренко Пётр Тихонович ", 1911.0,  
1106.66, 78, 8.17);  
} //FormCreate
```

В этом варианте приложения в конструкторе класса *TPrep* переопределены все поля наследуемых классов. Это позволяет при порождении экземпляра этого класса (в функции *FormCreate*) полям объекта назначить новые значения, не обращая внимания на то какие значения были присвоены полям при порождении этого класса или остались в наследство от классов-родителей.

14.7.3. Свойства конструкторов и правила их разработки

Ниже приведена в компактном виде уже упомянутая и дополнительная информация о конструкторах.

- ☐ Конструктор не возвращает значение.
- ☐ Имя конструктора совпадает с именем класса.
- ☐ Конструктор в классе должен объявляться со спецификатором доступа *public*, по умолчанию он является общедоступным методом.
- ☐ Конструкторы нельзя описывать с модификаторами *const*, *virtual* и *static*.
- ☐ Нельзя получить адрес конструктора, поэтому нельзя настроить указатель на конструктор.
- ☐ Может применяться *перегрузка конструкторов*. При её использовании класс имеет *несколько конструкторов* с разными параметрами, предназначенными для разных видов инициализации.
- ☐ Конструктор без параметров называется *конструктором по умолчанию*. Если класс не содержит ни одного конструктора, то автоматически создаётся конструктор, который также называется *конструктором по умолчанию*, он *только* выделяет память, необходимую для размещения экземпляра класса (инициализация полей не производится).
- ☐ Абстрактные классы не имеют конструкторов.
- ☐ *Конструктор с аргументами* позволяет по-разному инициализировать объект в момент его создания: вызывать различные функции, выделять динамическую память, присваивать переменным начальные значения и др.
- ☐ Параметры конструктора могут иметь *любой* тип, *кроме* этого же класса (но может быть ссылка на этот же класс).
- ☐ Можно задавать значения параметров по умолчанию, они могут содержаться *только у одного* из конструкторов.
- ☐ Конструкторы *не наследуются*. В производном классе *можно* вызывать конструкторы базовых классов.

14.8. Деструктор класса

14.8.1. Обычные деструкторы

Деструктором называется специальный метод класса, предназначенный для *освобождения памяти*, занимаемой объектом. В классе *всегда* имеется деструктор, он может быть скрытым или явным. *Явный* деструктор описывается программистом. Если деструктор явным образом *не определён*, *Builder* автоматически создаёт *скрытый* или *пустой деструктор*, который выполняет все возлагаемые на этот метод действия. Деструктор любого типа вызывается *автоматически* всякий раз, когда управление программой выходит из области видимости нединамического объекта. Имя деструктора должно совпадать с именем класса и предваряться тильдой «~» (без пробела или с одним и более пробелов). Деструктор не имеет параметров и возвращаемого значения.

Деструктор *следует явно описывать* в классе только в случае, когда порождаемый объект содержит поля, являющиеся динамическими переменными. В противном случае при уничтожении такого объекта динамическая память, на которую ссылаются его поля-указатели, не будет помечена как свободная, поэтому не сможет использоваться для *последующего* размещения переменных и объектов программы.

Деструкторы, как правило, выполняют операции обратные соответствующим конструкторам. Например, если конструктор класса с помощью операции *new* выделяет динамическую память для массива данных, то деструктор *освобождает* её с использованием операции *delete*. Но это вовсе не означает, что явные деструкторы могут иметься только у классов с явными конструкторами. Инициализацию полей-указателей класса способен выполнить и обычный метод, а возвращение памяти в свободный резерв кучи может осуществить деструктор. Может ли действия деструктора выполнить *обычный* метод класса? Конечно, может. Однако для этого его необходимо специально вызывать. Деструктор же вызывается *автоматически* при выходе управления программой из области видимости объекта. Поэтому применение деструкторов весьма удобно и способствует повышению надёжности программ.

Для иллюстрации деструкторов *простых* и *производных* классов изучим упрощённое приложение, которое аналогично приложению, рассмотренному в п. 14.7.2. Пусть имеется три класса. Базовый класс *A* наследуется классом *B*, класс *B* в свою очередь является родителем класса *C*. Ранее в этих классах были только целочисленные поля. Теперь же вместо них используем указатели на динамические массивы *различных типов*. В самом верхнем классе (в классе *A*) имеется только указатель на целочисленный динамический массив, в классе *B* к нему добавляется поле-указатель на динамический массив символов, а в классе *C* объявлено дополнительное поле - указатель на динамический массив строк.

Соответствующими конструкторами в целочисленный массив записываются числа от 1 до 5, в массив символов – буквы от *A* до *D*, а в массив строк – 5 женских

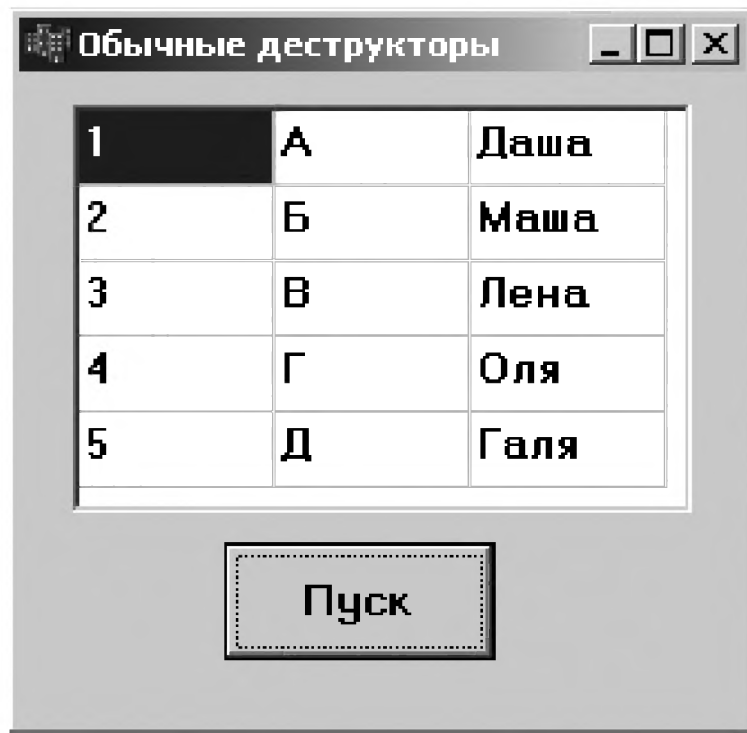


Рис. 14.1. Интерфейс приложения *Обычные деструкторы*

имён. На форме имеется только таблица строк и командная кнопка с заголовком *Пуск*. Форме и всем её визуальным объектам оставлены имена, присвоенные *Builder* по умолчанию. Интерфейс этого приложения (после нажатия кнопки *Пуск*) с заголовком *Обычные деструкторы* показан на рис. 14.1.

Для удобства чтения кода сразу же после объявления очередного класса описываются его методы. Конструкторы классов порождают указанные динамические массивы и инициализируют их упомянутыми выше значениями.

При чтении кода приложения обратите внимание на то, что в производных классах *не вызываются* деструкторы родителей, как казалось бы необходимо делать при разработке очередного дочернего класса. Дело в том, что эту операцию *Builder выполняет автоматически*, её программирование *не является* ошибкой, вместе с тем излишне. В случае наличия указанных деструкторов память, ранее выделенная под массивы разных типов данных, полностью возвращается системе при завершении работы приложения. При этом вначале вызывается деструктор самого сложного класса *С*, затем класса *В*, последним запускается деструктор класса *А*. Таким образом, очерёдность работы деструкторов иерархии классов строго *обратная* вызовам конструкторов соответствующих классов.

Конечно, во многих случаях при порождении динамических объектов *внутри* класса можно не применять явных деструкторов и при этом программы во многих случаях будут работать нормально, ведь не всегда же приложение оккупирует динамическую память критических объёмов (близкую к объёму свободной оперативной памяти). Однако, например, графические программы (они рассматриваются на следующем уроке) могут быстро исчерпать ресурсы памяти и ваша программа замрёт. Поэтому рекомендуем *всегда* применять *явные* деструкторы, если в классе используются динамические объекты. Теперь смотрите код приложения.

```

...
const int in= 5;

typedef int *Yk_int;
typedef char *Yk_char;
typedef String *Yk_String;

class A
{
    protected:
        Yk_int iMas;
    public:
        A(int inn);
        void Vuvod_Mas(int inn); //Вывод массива на экран
        ~A(); //Прототип деструктора класса A
}; //A

A::A(int inn) //Реализация конструктора класса A
{
    iMas= new int[inn];
    for (int ij= 0; ij< inn; ij++)
        iMas[ij]= ij + 1;
} //A::A

void A::Vuvod_Mas(int inn) //Вывод массива на экран
{
    for (int ij= 0; ij< inn; ij++)
        Form1->StringGrid1->Cells[0][ij]= iMas[ij];
} //A::Vuvod_Mas

A::~~A() //Деструктор класса A
{
    ShowMessage("Вызов деструктора A");
    delete [] iMas; //Освободили память, занятую массивом
    iMas= NULL; //В адресную часть iMas записали NULL или 0
} // A::~~A()

class B: public A
{
    protected:
        Yk_char cMas;
    public:
        void Vuvod_Mas(int inn); //Вывод массива на экран
        B(int inn):A(in)
        {
            cMas= new char[inn];
            for (int ij= 0; ij< inn; ij++)
                //Вводятся буквы А, Б, В, Г, Д
                cMas[ij]= (char) (ij + 192);
                // или cMas[ij]= (unsigned char) (ij + 192);
            } //B;
        ~B(); //Прототип деструктора класса B
}; //B

void B::Vuvod_Mas(int inn) //Вывод двух массивов на экран
{
    for (int ij= 0; ij< inn; ij++)

```

```

        {
            Form1->StringGrid1->Cells[0][ij]= iMas[ij];
            Form1->StringGrid1->Cells[1][ij]= cMas[ij];
        } //for_int
    } //B::Vuvod_Mas

B::~B() //Деструктор класса B
{
    ShowMessage("Вызов деструктора B");
    //A::~A(); //Деструктор класса A вызывается автоматически
    delete [] cMas; //Освободили память, занятую массивом
    cMas= NULL; //В адресную часть cMas записали NULL или 0
} //B::~B()

class C: public B
{
    protected:
        Yk_String sMas;
    public:
        void Vuvod_Mas(int inn);
        C(int inn):B(in)
        {
            sMas= new String[inn];
            sMas[0]= "Даша";
            sMas[1]= "Маша";
            sMas[2]= "Лена";
            sMas[3]= "Оля";
            sMas[4]= "Галя";
        }
        ~C(); //Прототип деструктора класса C
}; //C

void C::Vuvod_Mas(int inn)
{
    for (int ij= 0; ij< inn; ij++)
    {
        Form1->StringGrid1->Cells[0][ij]= iMas[ij];
        Form1->StringGrid1->Cells[1][ij]= cMas[ij];
        Form1->StringGrid1->Cells[2][ij]= sMas[ij];
    } //for_int
} //C::Vuvod_Mas

C::~C() //Прототип деструктора класса C
{
    ShowMessage("Вызов деструктора C");
    //B::~B(); //Вызывается автоматически
    delete [] sMas; //Освободили память, занятую массивом
    sMas= NULL; //В адресную часть cMas записали NULL или 0
} // C::~C()

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    //Устанавливаем число строк таблицы StringGrid1
    StringGrid1->RowCount= 5;
    //Другие настройки параметров StringGrid1
    StringGrid1->FixedRows= 0; //Нет заголовочной строки

```

```
StringGrid1->FixedCols= 0; //Нет заголовочного столбца
С Obj_C(in); //Порождён объект Obj_C класса С
Obj_C.Vuvod_Mas(in); //Вывод массивов iMas, cMas и sMas
} //Button1Click
```

Конструкторы классов *B* и *C* инициализируют не только поля, которые непосредственно в них описаны (соответственно *cMas* и *sMas*), но и наследуемые поля своих предков (соответственно *A* и *B*). Поэтому оператор *Obj_C.Vuvod_Mas(in);*, приведенный в функции *Button1Click*, наполняет данными *все три* столбца таблицы строк *StringGrid1*. Если вместо этого оператора привести оператор *Obj_A.Vuvod_Mas(in);*, то заполнен будет только первый столбец таблицы значениями *iMas*, использование оператора *Obj_B.Vuvod_Mas(in);* выводит в первые два столбца *StringGrid1* массивы *cMas* и *sMas* (см. рис. 14.1).

Для того, чтобы удостовериться в том, что в приведенном приложении *действительно* вызываются все три деструктора (по завершению работы функции *Button1Click*), причём в порядке обратном, порождению классов *A*, *B* и *C*, в тело *каждого* деструктора добавлена строка типа:

```
ShowMessage("Вызов деструктора класса B");
```

Имя класса в каждом деструкторе, конечно, своё. Порождение объекта *Obj_C* происходит не в файле реализации, а в теле функции *Button1Click* (этот объект является локальным для функции). Поэтому после нажатия кнопки *Пуск* на экране появится не только таблица строк с тремя заполненными столбцами, но и последовательно будут всплывать окна, информирующие о том, что произошёл вызов деструктора соответственно классов *C*, *B* и *A*. Эти окна появляются вследствие того, что при завершении работы функции, когда управление программой дойдёт до закрывающей её тело фигурной скобки, *Builder* автоматически вызывает перечисленные выше деструкторы, поскольку управление программой *выйдет* из *области видимости* объекта *Obj_C*. Если же порождение *Obj_C* перенести из функции *Button1Click* в файл реализации, то всплывающие окна с информацией о вызовах упомянутых деструкторов не выводятся, поскольку объект существует (живёт) вплоть до закрытия приложения. Обязательно проведите такой эксперимент.

14.8.2. Виртуальные деструкторы

В *C++* можно создавать *виртуальные деструкторы*. Все виртуальные методы класса, включая и виртуальные деструкторы, предназначены *только* для *динамических* объектов. В отличие от других виртуальных методов класса (которые в иерархии класса имеют *совпадающие* заголовки и прототипы), имена виртуальных деструкторов *совпадают* с именем класса, в котором они вводятся (поэтому в иерархии класса идентификаторы виртуальных деструкторов не одноимённые). Единственное синтаксическое отличие в оформлении класса с виртуальными деструкторами заключается лишь в том, что перед тильдой деструктора ставится служебное слово *virtual*. Код же реализации деструкторов не изменяется.

В теле деструктора каждого класса присутствует функция *ShowMessage* с сообщениями (например, сработал деструктор класса *C*), указывающими вызов конкретного деструктора. В этом приложении метод *Vuvod_Mas* также сделан виртуальным (все методы классов – виртуальные). Реализации методов *Vuvod_Mas* и всех деструкторов остались прежними, поэтому здесь не приводятся.

Цель настоящего приложения состоит иллюстрации возможности виртуальных деструкторов.

```
const int in= 5;

typedef int *Yk_int;
typedef char *Yk_char;
typedef String *Yk_String;

class A
{
    protected:
        Yk_int iMas;
    public:
        virtual void Inic(int inn); //Инициализация класса A
        virtual void Vuvod_Mas(int inn);
        virtual ~A(); //Виртуальный деструктор класса A
}; //A

void A::Inic(int inn)
{
    iMas= new int[inn];
    for (int ij= 0; ij< inn; ij++)
        iMas[ij]= ij + 1;
} //A::A_Inic

class B: public A
{
    protected:
        Yk_char cMas;
    public:
        virtual void Inic(int inn);
        virtual void Vuvod_Mas(int inn);
        virtual ~B(); //Виртуальный деструктор класса B
}; //B

void B::Inic(int inn)
{

```

```

    A::Inic(inn);
    cMas= new char[inn];
    for (int ij= 0; ij< inn; ij++)
        cMas[ij]= (char)(ij+ 192);
} //B_Inic;

class C: public B
{
protected:
    Yk_String sMas;
public:
    virtual void Vuvod_Mas(int inn);
    virtual void Inic(int inn);
    virtual ~C(); //Виртуальный деструктор класса C
}; //C

void C::Inic(int inn)
{
    B::Inic(inn);
    sMas= new String[inn];
    sMas[0]= "Даша";
    sMas[1]= "Маша";
    sMas[2]= "Лена";
    sMas[3]= "Оля";
    sMas[4]= "Галя";
} //C_Inic

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    A *Yk_Kl_A; //Указатель на базовый класс A
    //В указатель Yk_Kl_A записывается адрес объекта класса C
    Yk_Kl_A= new C;
    Yk_Kl_A->Inic(in); //Вызывается метод класса C

    //Устанавливаем число строк и столбцов таблицы StringGrid1
    StringGrid1->RowCount= 5;
    StringGrid1->ColCount= 3;
    //Другие настройки параметров StringGrid1
    StringGrid1->FixedRows= 0; //Нет заголовочной строки
    StringGrid1->FixedCols= 0; //Нет заголовочного столбца
    Yk_Kl_A->Vuvod_Mas(in); //Вывели массивы iMas, cMas и sMas
} //Button1Click

```

В этом приложении объявляется указатель *Yk_Kl_A* на класс *A* (общий родитель классов *B* и *C*). Однако далее этому указателю присваивается адрес динамического объекта класса *C*. Поскольку методы инициализации *Inic* *всех* классов *A*, *B* и *C* являются виртуальными, поэтому оператор *Yk_Kl_A->Inic(in)*; вызывает метод *Inic* класса *C*, а не класса *A*. В результате *все* переменные класса *C* будут инициализированы и, как следствие, столбцы таблицы строк будут заполнены данными (см. рис. 14.1). Такое заполнение таблицы данными, помещённых в объект при его инициализации, осуществляет виртуальный метод *Vuvod_Mas*. При использовании не виртуального метода *Inic* заполненным оказался бы *только* первый столбец таблицы.

Для динамических объектов сложной иерархии (с динамическими полями) рекомендуется *всегда* создавать виртуальные деструкторы. Однако в C++ при выходе из области видимости указателя на динамический объект *автоматический* вызов его деструкторов *не производится*. Доказательством этого является то, что в приведенном выше приложении информация о вызовах деструкторов на экран *не выводится*. Поэтому деструктор следует вызывать *явным образом* путём указания полностью его уточнённого имени. В обсуждаемом приложении для *корректного* освобождения памяти, занятой динамическими массивами, в самом конце кода функции *Button1Click* осуществим *явный* вызов виртуального деструктора:

```
Yk_Kl_A->~A();
```

В результате на экране последовательно появятся сообщения: сработал деструктор класса *C*, сработал деструктор класса *B*, сработал деструктор класса *A*.

Послушайте, возразит нам дотошный читатель, но ведь виртуальный деструктор класса *C* следует вызывать таким кодом?:

```
Yk_Kl_A->~C();
```

Однако в этом случае *на стадии компиляции* программы выводится сообщение об ошибке: *Destructor name must match the class name*. Оно свидетельствует о том, что среда программирования требует соответствия имени класса, на который объявлен указатель *Yk_Kl_A* имени его деструктора *~A()*. При этом как бы игнорируется тот факт, что указателю *Yk_Kl_A* присвоен адрес объекта класса *C*. Однако код *Yk_Kl_A->~A();* осуществляет вызов деструкторов родительского класса и предков в *правильной* последовательности: *C*, *B* и *A*. Столь нелогичный синтаксис виртуальных деструкторов не описан в литературе, поэтому автору все его особенности пришлось отвоёвывать у *Builder* экспериментальным способом.

14.8.3. Свойства деструкторов и правила их разработки

- ☐ Имя деструктора начинается с тильды, за которой следует имя класса.
- ☐ Деструкторы не имеют аргументов и возвращаемого значения (даже тип *void*).
- ☐ Деструкторы не наследуются. Если программист не описал в дочернем классе деструктор, он формируется по умолчанию и автоматически вызывает деструкторы всех базовых классов.
- ☐ Явные деструкторы дочерних классов автоматически вызывают деструкторы базовых классов. Поэтому вызов в дочерних классах деструкторов родительских классов является излишним, но синтаксисом разрешено.
- ☐ Деструкторы нельзя перегружать.
- ☐ Деструкторы не могут объявляться со служебным словом *const* или *static*.
- ☐ По умолчанию деструкторы создаются как общедоступные.

- ❑ Для нединамических объектов деструкторы вызывается автоматически, когда управление программой выходит из области видимости объекта.
- ❑ Для указателей на динамические объекты деструкторы автоматически не вызываются при выходе из области видимости таких объектов. Поэтому для указанных объектов деструкторы вызываются явным способом.
- ❑ Если деструктор явно не описан, автоматически создаётся пустой деструктор.
- ❑ Описывать деструктор явно следует в случае, когда объект имеет динамические поля-переменные.

14.9. Создание копий объектов

14.9.1. Побитовое копирование

У каждого класса может быть *несколько* конструкторов с разными параметрами и без них: может осуществляться перегрузка конструкторов. В этом случае в зависимости от числа, типа и очерёдности типов параметров, указанных при порождении объекта, вызывается соответствующий конструктор.

Среди конструкторов класса могут иметься не только конструкторы, предназначенные для *инициализации* полей экземпляров класса, но и так называемые *конструкторы копирования*. Если программист не укажет ни одного конструктора копирования, то он создаётся *автоматически*. Такой конструктор выполняет побитовое копирование полей одного объекта в другой. При этом типы этих объектов должны быть тождественными либо совместимыми.

Имеется *пять* вариантов синтаксиса операции побитового копирования. Покажем их на примере простого класса с инициализирующим конструктором по умолчанию (копирование сложных классов выполняется аналогичным образом):

```
class A
{
    public:
        int ia;
        A(int iaa= 0);
}; //A

A::A(int iaa)
{
    ia= iaa;
}; //A::A
```

Пусть на форме находится только командная кнопка с именем *Button1*. В функции *Button1Click* объявим объект *Obj_A* класса *A* с параметром 3 для инициализации поля *ia*.

```
A Obj_A(3);
```

Ниже запишем строку:

```
ShowMessage(Obj_A.ia); //3
```


Этот оператор выведет на экран значение инициализированного поля. При инициализации по умолчанию

```
A Obj_A;
```

на экране появится уже не число три, а нуль.

Скопировать все поля объекта *Obj_A* в поля объекта *Obj_AA* этого же класса *A* можно при помощи следующего оператора копирования:

```
A Obj_AA= Obj_A;
```

В нашем примере в каждом из объектов имеется только одно поле *ia*. Если бы их было больше, то все их значения скопируются из объекта-источника (в нашем примере *Obj_A*) в соответствующие поля объекта-копии (в нашем примере *Obj_AA*). Причём такое копирование является *побитовым* или *поэлементным*.

Теперь функция

```
ShowMessage(Obj_AA.ia); //0
```

выведет на экран значение поля *ia* в объекте *Obj_AA*. В полях *ia* указанных объектов находится одно и то же значение. В дальнейшем значения этих полей могут изменяться *независимо* друг от друга.

Имеется *ещё четыре* разновидности синтаксиса оператора копирования:

```
A Obj_AA= A(Obj_A); //вариант 2
```

```
A Obj_AA (Obj_A); //вариант 3
```

```
A Obj_AA= A(5); //вариант 4, на экране покажется число 5
```

```
A Obj_AA= 5; //вариант 5, на экране покажется число 5
```

В варианте 4 создаётся *безымянный* объект типа *A* со значением поля *ia* == 5. После этого выделяется память под объект *Obj_AA* такого же типа *A*, в этот блок памяти копируется безымянный объект с инициализированным полем *ia*. Если в классе *A* имеется несколько полей, то в ходе создания его безымянного объекта *в общих* круглых скобках можно указать значение *всех* полей (перечислив их через запятую), либо некоторые из них. В последнем случае неуказанные значения полей (описанные последними в классе) устанавливаются по умолчанию.

В варианте 5 происходят те же действия, что и в варианте 4. Однако если у объектов будет несколько параметров, то таким способом можно инициализировать только *первый* из них, все другие устанавливаются *лишь* по умолчанию. Поэтому последний способ можно рекомендовать в случае, когда для инициализации объекта допускается задавать только первый параметр.

Ниже используется только *первый* вариант синтаксиса операции копирования объектов, поскольку на наш взгляд он является наиболее ясным.

14.9.2. Запрет неявного преобразования типов

В пятом варианте копирования имеется возможность реализовать запрет неявного преобразования типа. Напоминаем, что в *C++* по умолчанию может происходить *неявное* преобразование типа. Например, если во всех 5 способах копирования объекта поле *ia* (типа *int*) инициализировать вещественным значением, то в нём записывается только целая часть числа: произойдёт неявное преобразование инициализирующего значения к типу *int*. Программисты всегда должны иметь это в виду. В тех случаях, когда требуется защитить программу, например

от некорректных значений, введенных пользователем, можно применить последний способ копирования: он позволяет запретить неявное преобразование типа (конечно, только в упомянутой операции копирования). С этой целью в классе перед объявлением конструктора указывается служебное слово *explicit* (явный):

```
explicit A(int iaa= 0);
```

Теперь при действительном значении инициализирующего значения, например:

```
A Obj_AA= 5.8;
```

компиляция останавливается и выводится сообщение о невозможности преобразования (в нашем примере *double* в класс *A*). При этом даже явное преобразование типа

```
A Obj_AA= int(5.8);
```

не позволит продолжить дальнейшую компиляцию. Наличие такого ограничения является некоторым преимуществом последнего способа. Однако на наш взгляд этот способ очень уж неясный (и неудобный) и, несмотря на указанное преимущество, не рекомендуем его применять.

14.9.3. Конструктор копирования

Напоминаем, в случае, когда конструктор копии не объявлен, то при всех рассмотренных вариантах копирования выполняется *побитовое копирование* полей из одного объекта в другой. При побитовом копировании для простых типов полей не возникает никаких проблем. Однако динамические поля при побитовом копировании ведут себя неправильно, поскольку указатель копии указывает на данные источника. Поэтому копия и оригинал в этом случае не являются независимыми объектами. Все изменения поля-указателя копии происходят и в том же поле-указателе источника и наоборот.

Для иллюстрации этого эффекта приведём программу, в которой в единственном классе *A* имеется динамический целочисленный массив *iMas*, который создаётся в конструкторе, а инициализируются методом *Vvod*, методом *Vuvod_Mas* он выводится на экран в таблицу строк *StringGrid1*. Метод *Vvod* имеет целочисленные параметры *in* и *ia*, которые позволяют менять длину и значения элементов массива *iMas*. Для индикации запуска пустого конструктора в нём при помощи функции *ShowMessage* выводится соответствующее сообщение.

```
...
const int in= 5;
typedef int *Yk_int;

class A
{
    protected:
        Yk_int iMas;
    public:
        A(void); //Конструктор
        void A::Vvod(int inn, int ia );
        void Vuvod_Mas(int inn);
```

```

    ~A();
}; //A

A::A(void) //Конструктор
{
    iMas= new int[in];
    ShowMessage("Запущен конструктор класса A ");
} //A::A

void A::Vvod(int inn, int ia )
{
    for (int ij= 0; ij< inn; ij++)
        iMas[ij]= ij + ia;
} //A::Vvod

void A::Vuvod_Mas(int inn)
{
    for (int ij= 0; ij< inn; ij++)
    {
        Form1->StringGrid1->Cells[0][ij]= iMas[ij];
    } //for_int
} //A::Vuvod_Mas

A::~A() //Деструктор класса A
{
    ShowMessage("Сработал деструктор класса A ");
    delete [] iMas;
    iMas= NULL;
} //A::~A()

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    StringGrid1->RowCount= 5;
    StringGrid1->ColCount= 1;
    StringGrid1->FixedRows= 0;
    StringGrid1->FixedCols= 0;

    A Obj_Kl_A; //Создан объект Obj_Kl_A класса A
    Obj_Kl_A.Vvod(in, 0); // Ввод массива
    Obj_Kl_A.Vuvod_Mas(in);
    ShowMessage("Задержка в работе программы");
    A Obj_Kl_A_1= Obj_Kl_A; //Создается побитовая копия объекта
    Obj_Kl_A_1.Vuvod_Mas(in);
    ShowMessage("Задержка ");
    Obj_Kl_A_1.Vvod(in, 5); //Ввод новых значений массива
    //ShowMessage("Задержка");
    Obj_Kl_A_1.Vuvod_Mas(in);
    ShowMessage("Работа программы задержана");
    Obj_Kl_A.Vuvod_Mas(in);
} //Button1Click

```

При начальной инициализации массиву *iMas* в объекте *Obj_Kl_A* назначены значения от 1 до 5. Эти значения выводятся методом *Vuvod_Mas* как из объекта *Obj_Kl_A*, так и объекта *Obj_Kl_A_1*. Далее этому же массиву, но в объекте-копии *Obj_Kl_A_1* присваиваются новые значения элементов (от 5 до 9), после чего методом *Vuvod_Mas* из объектов *Obj_Kl_A_1* и *Obj_Kl_A* выводится числа от 5 до 9.

Таким образом, изменения массива-копии произошли и в массиве-оригинале, поскольку указатель динамического массива копии и оригинала указывает на одну и ту же ячейку памяти – начальную ячейку массива источника. При этом следует особо подчеркнуть, что в программе конструктор вызывается только один раз – при *создании* объекта *Obj_Kl_A* (при этом выводится сообщение «*Запущен конструктор класса А*»). При побитовом *копировании* (оно осуществляется при помощи автоматически сгенерированного конструктора, либо с использованием конструктора без параметров) конструктор не вызывается. В результате уничтожения объектов *Obj_Kl_A_1* и *Obj_Kl_A* дважды вызывается деструктор класса, поэтому сообщение «*Сработал деструктор класса А*» выводится два раза.

Для того чтобы оригинал и копия объекта были независимы, можно поступить двумя способами. Первый из них состоит в *переносе* строки кода

```
iMas= new int[in];
```

из конструктора в метод *Vvod* (в качестве первой строки его тела). В этом случае перед *каждой* инициализацией массива *iMas* ему в куче выделяется *новый* адрес.

Второй способ состоит в применении *конструктора копирования*. Это специальный вид конструктора, получающий в качестве *единственного* параметра адрес (ссылку) *оригинала* объекта. Классы копии объекта и оригинала, конечно, должны быть тождественными. Покажем работу конструктора копирования на примере прежнего приложения. В классе осуществим перегрузку конструкторов. Оставим прежний конструктор класса *A* без изменения, он будет использоваться при создании объекта-оригинала *Obj_Kl_A*. Второй конструктор – конструктор копирования:

```
A(const A & Objekt);
```

Его единственный параметр – константная ссылка на объект этого же класса *A*. Благодаря различию параметров упомянутых конструкторов (в первом нет параметров, а во втором – только один) первый конструктор вызывается при создании объекта, а второй – при создании его копии. Очередность их расположения в классе не имеет значения. Посмотрите код этого конструктора.

```
A::A(const A & Objekt)
{
    ShowMessage("Вызван конструктор копирования класса А ");
    iMas= new int[in]; //Получен новый адрес для iMas
    for (int ij= 0; ij< in; ij++)
        iMas[ij]= Objekt.iMas[ij]; //В новые ячейки массива
                                   //копируются значения ячеек оригинала
} //A::A
```

В конструкторе копии порождается *новая* ссылка на динамический массив, далее осуществляется поэлементное копирование значений из массива-оригинала в массив-копию.

Все другие фрагменты прежнего приложения остались без изменения. Теперь исходный объект *Obj_Kl_A* и его копия *Obj_Kl_A_1* полностью *независимы*. В ходе работы функции *Button1Click* дополнительно выводится сообщение при создании копии объекта *Obj_Kl_A* (то есть объекта *Obj_Kl_A_1*): «*Вызван конструктор копирования класса А*».

14.10. Указатель *this*

14.10.1. Назначения и возможности указателя *this*

Ключевое слово *this* (этот) является *рекурсивным константным обобщённым* указателем, который *неявно* объявляется в *каждом* классе для хранения в нём адреса порождаемого экземпляра класса. Этот указатель является обобщённым, поскольку тип класса может быть произвольным. В классе это скрытое от программиста поле объявляется так:

```
Primer * const this; //Primer — это имя произвольного класса
```

Указатель *this* — это константный указатель на *неконстантный* объект: адрес, записанный в указателе, не меняется, а сам объект может изменяться — полям экземпляра класса можно присваивать различные допустимые значения.

При порождении *каждого* объекта этот указатель *автоматически* инициализируется адресом порождаемого объекта. Именно поэтому он называется *рекурсивным*: в нём содержится адрес объекта, в котором он сам же и находится. Поскольку упомянутый указатель является *неявным*, то в пользовательских и стандартных (библиотечных) классах он *невидим*; объявлять его программисту самостоятельно *нельзя*, поскольку *Builder* выполняет эту работу *самостоятельно*.

Покажем пример использования этой рекурсивной невидимки. При запуске нижеследующего приложения на экране появляется только одна буква. Класс описан в файле реализации, а объявление объекта, его инициализация и вывод на экран значения единственного поля осуществляются в функции *FormCreate*.

```
class Primer
{
    char Ch; // Закрытое поле класса
    public:
        void Inic(char Chr){this->Ch= Chr;} //Метод записи
        char Vuvod(void) {return this->Ch;} //Метод чтения
}; //Primer

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Primer Prim; //Объявлен объект
    Prim.Inic ('Я'); //Запись в поле выбранной литеры
    ShowMessage(Prim.Vuvod()); //Вывод значения закрытого поля
} //FormCreate
```

В данном примере указатель *this* используется для доступа к полю *Ch* объекта *Prim* через указатель на этот объект. Конечно, к единственному полю объекта *Prim* в методах *Inic* и *Vuvod* класса *Primer* можно обратиться и непосредственно (через скрытый указатель):

```
void Inic(char Chr){Ch= Chr;}
char Vuvod(void) {return Ch;}
```

Опишем теперь механизм *неявного* доступа указателя *this* к методам класса. Каждый *метод* класса имеет *скрытый* первый параметр, при порождении объек-

та этот параметр *автоматически* инициализируется указателем *this* – адресом созданного объекта, поле которого должен обрабатывать вызываемый метод.

Поэтому каждый объект имеет доступ к собственному адресу с помощью указателя *this*. Казалось бы, этот указатель *должен* являться частью самого объекта. Однако операция *sizeof(Prim)* для объекта (в примере *Prim*) место под указатель *не показывает*. Аналогичная ситуация наблюдается для типов *char** и *String*, когда операция *sizeof* не показывает место для завершающего нулевого символа строки. При этом заметим, что в *каждом* стандартном типе данных хранится информация не только об объёме *переменной* выбранного типа, но и скрытые данные о её устройстве. Это и позволяет на этапе компиляции программы бить тревогу, когда в переменную назначаются данные того же объёма, что и её тип, однако тип назначаемых переменных (или данных) не родственен типу принимающей переменной.

Второй пример услуг среды программирования: при определении размера класса операцией *sizeof* получите объём *экземпляра класса* (объекта), а не класса, в котором хранятся только методы.

В предшествующих примерах без указателя *this* можно обойтись. Модернизируем теперь прежнюю программу таким образом, чтобы проиллюстрировать один из случаев, когда без этого указателя обойтись нельзя. С этой целью сделаем так, чтобы имя формального параметра в методе записи *Inic* было таким же, как и имя поля класса. Конечно, такое совпадение идентификаторов следует избегать, однако ниже используем образец такого плохого стиля программирования только для иллюстрации возможностей изучаемого указателя.

```
void Inic(char Ch){this->Ch= Ch;}
```

Если в этом методе не использовать указатель *this*, то у *Builder* нет возможности разобраться, где поле объекта, а где формальный параметр метода и поэтому на экран буква *Я* не выводится.

Вместо указателя *this* можно применить также операцию уточнения (указания) области видимости:

```
void Inic(char Ch){Primer::Ch= Ch;}
```

В этом случае на экране вновь появится прежняя буква.

14.10.2. Программное порождение визуальных объектов

Ранее во всех программах только форма появлялась на экране автоматически (без участия программиста). Все другие объекты, принадлежащие библиотеке *VCL* (как видимые, так и невидимые), порождались в *ходе разработки* приложения с использованием *Палитры Компонентов*. Теперь овладеем способом *программного* порождения визуальных объектов. Проиллюстрируем его на очень простых примерах.

Приложение, код которого показан ниже, состоит лишь из единственной командной кнопки *Button1* на форме *Form1*. При нажатии этой кнопки в центре

формы генерируется поле вывода (с именем *Label*), внутри которого записывается число, равное текущему значению *ширины* формы. После изменения мышью ширины формы и последующего нажатия командной кнопки выводится *новая* ширина в *новом* центре формы. Кнопку расположите подальше от центра формы, её габаритные размеры сделайте небольшими с тем, чтобы она не заслоняла сообщение, выводимое в метке *Label*.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TLabel *Label= new TLabel(this);
    Label->Parent= this;
    Label->Left= this->Width/2;
    Label->Top= this->Top/2;
    Label->Caption= this->Width;
} //Button1Click
```

Если требуется какой-либо объект *VCL* создать динамически (в процессе выполнения программы), то конструктору его класса необходимо передать фактический параметр – указатель *this* формы. В первой строке кода функции *Button1Click* неявный указатель формы *this* передаётся конструктору класса *TLabel*:

```
TLabel *Label= new TLabel(this);
```

Конструктор выполняет присваивание значения переданного параметра *this* свойству *Owner* (владелец) метки *Label*. Теперь метка *Label* знает, *кто* является её владельцем, внутри какого объекта она расположена. Более того, и форме теперь известно, что ей принадлежит метка *Label*. Поэтому, когда форма завершает свою работу (при закрытии приложения), автоматически уничтожается и указанное поле вывода.

При порождении метки (например, *Label1* путём двойного щелчка по её заготовке в *Палитре Компонентов*) в классе формы появится новое поле в разделе *__published*:

```
TLabel *Label1;
```

В этом легко убедиться: посмотрите класс формы после порождения метки (команда *Ctrl+F6*). Когда форма (в нашем примере *Form1*), владеющая компонентом *Label*, уничтожается (завершает свою работу), автоматически уничтожается и указанное поле вывода. Такое автоматическое уничтожение всех объектов формы осуществляется только в случаях, когда объекты порождены с использованием *Палитры Компонентов*, либо способом, указанным выше (с применением указателя *this*).

Однако программно порождаемому объекту формы можно указать также и нулевой адрес (или *NULL*-адрес):

```
TLabel *Label= new TLabel(NULL); //Глобальный указатель
```

В этом случае такой объект исчезающей формой *не уничтожится*. Работу по его уничтожению (освобождению памяти, занимаемой объектом) перед завершением работы формы следует предусмотреть в программе, например при помощи командной кнопки с именем *Vuxod*:

```
void __fastcall TForm1::VuxodClick(TObject *Sender)
{
    delete Label;
    Close();
} //VuxodClick
```

Чтобы визуальный компонент *прорисовался* на форме (оказался видимым), его свойству *Parent* (родитель, предок) нужно присвоить указатель *this* родителя-контейнера – объекта, внутри которого содержится новый компонент. Это свойство в *Инспекторе Объектов* не отражается. Любой визуальный объект можно расположить также *внутри* различных панелей или контейнерных классов. Вместе с тем, все они будут принадлежать и форме. После инициализации поля *Parent* указателем *this* родительского объекта (в нашем примере *Form1*) компонент *Label* *наследует* метод прорисовки объекта (и многие другие методы, но не все), в результате чего объект становится *видимым*. Такое присваивание указателя *this* и прорисовка объекта на стадии проектирования приложения с использованием *Палитры Компонентов* происходит *автоматически*. В случае же *программного* порождения объекта следует *явно* записать код присваивания, иначе компонент не будет отображён (см. вторую строку кода функции *Button1Click*).

```
Label->Parent= this;
```

Во всех упомянутых случаях указатель *this* направлен на форму *Form1*. Поэтому если во всех (или только в выбранных) строках кода функции *Button1Click* идентификатор *this* заменить именем формы *Form1*, то приложение будет работать точно также как и ранее, ведь *Form1* (или любое другое имя, которое выбрано для формы) – это также указатель, который объявляется *автоматически* и направляется на *автоматически* порождённую форму.

Поэтому каждая из строк

```
Label->Parent= this;
Label->Parent= Form1; //Более ясный способ
```

определяет принадлежность метки *Label* форме *Form1*.

Последующие строки кода функции *Button1Click* (3, 4 и 5 строки) определяют центр формы по вертикали и горизонтали, располагают в нём верхний левый угол метки *Label*, после чего в этом поле вывода записывается текущая ширина формы.

В следующем приложении используем тело конструктора формы, генерируемое *Builder* автоматически. Внутри этого тела запишем строки кода, которые при запуске приложения программно выводят на форму таблицу строк с заданными свойствами: в её ячейки можно вводить данные и переходить из одной ячейки в другую при помощи клавиши *Tab*.

```
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
    TStringGrid *StringGrid= new TStringGrid(this);
    StringGrid->Parent= this;
    StringGrid->Options<<goEditing<<goTabs;
} //TForm1::TForm1
```


Также как и в предшествующем приложении, здесь служебное слово *this* можно заменить идентификатором формы *Form1*. Возможность ввода данных в ячейки таблицы и применение клавиши *Tab* для перемещения по её ячейкам осуществляется активизацией опций *goEditing* и *goTabs*, принадлежащих свойству *Options*. Указанные опции в свойстве *Options* описаны как множества, поэтому для их программного включения используется знак «<<» помещенный во множество. При помощи *Инспектора Объектов* указанные опции устанавливаются выбором значения *true* (вместо значения *false*, установленного по умолчанию). Выключение опций осуществляется при помощи знака «>>» – исключение из множества, например:

```
StringGrid->Options>>goEditing;
```

При компиляции последней программы получите сообщения «*Undefined symbol TstringGrid, symbol StringGrid*» и другие ошибки. Такая неадекватная реакция компилятора указывает на недостатки (на жаргонном языке – глюки) приложений *Builder 5.0* и *Builder 6.0* (а также и *Builder 2009*). Они наблюдаются и при порождении некоторых других визуальных компонентов. Обойти указанные несовершенства компилятора можно так:

- ❑ С использованием *Палитры Компонентов* породить на форме таблицу строк, например, *StringGrid1* (в нашем примере).
- ❑ Откомпилировать программу.
- ❑ Уничтожить объект, порождённый при помощи *Палитры Компонентов* (*StringGrid1*).
- ❑ Вновь откомпилировать приложение.

В результате этих действий приложение будет порождать упомянутый визуальный объект.

В следующем приложении на форме *Form1* вначале имеется только командная кнопка с именем *Pysk* (её заголовок *Пуск*). Она порождается на стадии проектирования программы с использованием *Инспектора Объектов*. В функции *PyskClick* программно генерируется метка *Label*, в заголовке которой отображается значением глобальной переменной *iSchetchik* (номер очередного нажатия кнопки *Пуск*). При чётных и нечётных значениях переменной *iSchetchik* местоположения порождаемой метки различны. Функция *PyskMouseDown* обрабатывает событие *нажатие командной кнопки*. Эта функция стирает (записывает строку, состоящую из пяти пробелов) заголовок метки перед каждым щелчком по командной кнопке. В результате поочередной работы функций *PyskMouseDown* и *PyskClick* изображение номера очередного нажатия кнопки осуществляет прыжки по горизонтали на треть текущей ширины формы.

```
...
TLabel *Label;//Глобальный указатель
int iSchetchik= 0;//Число нажатий

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    iSchetchik++;//Счётчик нажатий
```

```

Label= new TLabel(this);
Label->Parent= this;
//Вертикальная координата кнопки не изменяется
Label->Top= Form1->Height/2;
if (iSchetchik%2== 0)//Чётные значения iSchetchik
{
    Label->Left= Form1->Width/3;
    Label->Caption= iSchetchik;
} //if
else//Нечётные значения iSchetchik
{
    Label->Left= Form1->Width/3*2;
    Label->Caption= iSchetchik;
} //else
} //Button1Click

void __fastcall TForm1::PyskMouseDown(TObject *Sender, TMouseButton Button,
                                     TShiftState Shift, int X, int Y)
{
    Label= new TLabel(this); //Нельзя разместить
    Label->Parent= this; //вне функции
    Label->Top= Form1->Height/2;

    if (iSchetchik%2== 0)
    {
        Label->Left= Form1->Width/3;
        Label->Caption= "    ";
    } //if
    else
    {
        Label->Left= Form1->Width/3*2;
        Label->Caption= "    ";
    } //else
} //PyskMouseDown

```

В этой программе все вхождения служебного слова *this* можно заменить указателем *Form1* и наоборот имя формы *Form1* допустимо заменить указателем *this*.

14.11. Локальные классы

Класс можно определить внутри *любого* логического блока, например, внутри функции. Такой класс называется *локальным*, он *недоступен* вне функции или блока, где описан. Методы локальных классов (как и структур) должны описываться *только* внутри класса.

В локальном классе не могут объявляться статические данные, поскольку его поля и методы видимы только внутри блока, в котором он содержится. Внутри локального класса разрешается использовать глобальные для него типы и переменные (включая статические и внешние, описываемые со спецификаторами *static* и *extern*), функции и элементы перечислений. Проиллюстрируем применение локального класса в приложении, в файле реализации которого опишем глобальный тип и переменную:

```
typedef int Tip_int;  
extern Tip_int iy= 2;
```

Код, приведенный ниже, можно разместить в произвольной пользовательской функции, конструкторе приложения или, например, в функциях *FormCreate* и *Button1Click*.

```
class Lokaln_Klass  
{  
    int ix;  
public:  
    void Zapisj(int ixx){ix= ixx;}  
    int Pokaz(void){++iy; return ix + iy;}  
}Lok_Kl;  
Lok_Kl.Zapisj(5);  
ShowMessage(Lok_Kl.Pokaz());
```

Поместим указанный код в функции *Button1Click*. В результате щелчка по командной кнопке на экране появится число 8, при последующих её нажатиях выводятся числа 9, 10 и т.д. Полагаем, что локальные классы не находят широкого применения.

14.12. Шаблоны классов

Шаблоны классов и шаблоны функций выполняют *очень сходные* задачи. Механизм шаблонов и в том и другом случае позволяет генерировать *целое семейство* подобных кодов (соответственно функций и классов), которые отличаются друг от друга *только типом* данных. Шаблоны дают возможность отделить алгоритм от *конкретных* типов данных, они могут применяться к любым допустимым в алгоритме типам данных без переписывания кода. При использовании шаблона функций *не требуется* указывать тип ни одного параметра шаблона: шаблонная функция генерирует типы всех своих аргументов по типам фактических параметров применённых при её вызове. В случае же шаблонов классов тип параметра следует *явно* указывать: в этом заключается *единственное* отличие в их возможностях.

Шаблоны классов называют также родовыми (*generic*) классами или генераторами классов. Менее часто их именуют параметризованными типами, поскольку они имеют один или более параметров типа их полей, определяющих настройку шаблона класса на конкретные типы данных при создании объекта класса. Все перечисленные разновидности названий вполне логичны и поэтому продолжают использоваться в литературе. Применение шаблонов сокращает объём приложения: программист описывает шаблон только *один* раз, во всех же случаях, когда требуется реализация класса для *нового* типа данных, в приложении используется краткая запись, позволяющая *автоматически* создать экземпляр требуемого класса.

Синтаксис описания шаблонов и операции порождения (с их помощью) экземпляров класса рассмотрим на примере упрощённого учебного приложения.

Разработаем шаблон класса, порождающий двумерный массив с формальным типом элементов. Пусть все элементы матрицы инициализируются конструктором при её порождении. Помимо конструктора существуют также методы записи элементов матрицы и их вывода в таблицу строк, расположенную на интерфейсе приложения. Порождение таблицы строк и экземпляра класса производится в функции *Button1Click*.

В файле реализации опишем шаблон класса с именем *Matrica*.

```
template <class T> class Matrica
{
    T *Tx; //Содержимое ячеек матрицы имеет формальный тип T*
    int in_St; // Число строк
    int in_Cl; //Число столбцов
public:
    Matrica(int in_Str, int in_Col, T Txx); //Конструктор
    ~Matrica(){delete [] Tx;} //Деструктор
    void Zapisj (int in_Str, int in_Col, T Txx);
    void Chtenie (TStringGrid *StringGr, int in_Str, int in_Col);
}; //Matrica
```

В этом описании использовано служебное слово *template* (шаблон). В угловых скобках указывается формальный тип переменных, помещаемых в элементы матрицы: `<class T>`. По негласному соглашению для обозначения этого типа применяется прописная буква *T*. Вместе с тем можно использовать, например, такой идентификатор *Tip* (или любой другой). Ключевое слово *class* – очень неудачный элемент синтаксиса, поскольку типом данных может быть не только класс, но и любой другой тип (например, *int*, *float*, *double*, *char*, *String* и др. стандартные и пользовательские типы, включая структуры и классы). В *C++Builder 6.0* вместо этого спецификатора можно задавать ключевое слово *typename*:

```
template <typename T> class Matrica
```

В этом случае смысл параметра *T* становится более ясным.

Ниже приведём вариант реализации конструктора. Обратите внимание на его заголовок: перед знаком операции расширения области видимости требуется указывать элементы заголовка шаблона класса.

```
template <class T> Matrica<T>::Matrica(int in_Str, int in_Col, T Txx)
{
    Tx= new T[in_Str*in_Col];
    for (int ii= 0; ii< in_Str; ii++)
        for (int ij= 0; ij< in_Col; ij++)
            Tx[ii*in_Col + ij]= Txx;
    in_St= in_Str;
    in_Cl= in_Str;
} //Matrica::Matrica
```

Поскольку массив является динамическим, то в первой строке кода в указатель *Tx* на тип *T* записывается адрес начальной ячейки одномерного массива, число ячеек которого равно числу ячеек требуемого двумерного массива. Применение одномерного динамического массива вместо двумерной матрицы

значительно удобнее. В п. 10.7.2 описываются громоздкие операции порождения и уничтожения двумерного динамического массива. Однако в памяти компьютера массив *любой* размерности представляется в виде *одномерного* массива: вначале записываются ячейки первой строки, затем – второй третьей и так вплоть до последней, далее в аналогичном порядке записываются строки двумерного массива следующего измерения и так далее. Поэтому в нашем случае, зная местоположение каждой ячейки по индексам её строки и столбца, легко записать в неё требуемую информацию, а затем считать её в выбранную переменную.

Устройство методов *Zapisj* и *Chtenie*, предназначенных соответственно для обновления данных матрицы и для вывода их в таблицу строк *StringGr* аналогично устройству конструктора.

```
template <class T> void Matrica<T>::Zapisj (int in_Str, int in_Col, T Txx)
{
    for (int ii= 0; ii< in_Str; ii++)
        for (int ij= 0; ij< in_Col; ij++)
            Tx[ii*in_Col + ij]= Txx;
} //Matrica<T>::Zapisj

template <class T> void Matrica<T>::Chtenie (TStringGrid
                                           *StringGr, int in_Str, int in_Col)
{
    for (int ii= 0; ii< in_Str; ii++)
        for (int ij= 0; ij< in_Col; ij++)
            StringGr->Cells[ij][ii]= Tx[ii*in_Col + ij];
} //Matrica<T>::Chtenie
```

В функции *Button1Click* осуществляется порождение таблицы строк *StringGrid* и настройка её элементов. В предпоследней строке кода этой функции порождается объект *iM* требуемого класса – матрица с заданным числом строк, столбцов и типом *int* её элементов. При порождении матрицы осуществляется инициализация всех её ячеек числом 12. Последней строкой кода функции *Button1Click* значения элементов матрицы выводятся на экран.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    const int inStr= 3, inCol= 4;
    TStringGrid *StringGrid= new TStringGrid(this);
    StringGrid->Parent= this;
    StringGrid->RowCount= inStr;
    StringGrid->ColCount= inCol;
    StringGrid->FixedRows= 0;
    StringGrid->FixedCols= 0;
    //Порождение объекта iM при помощи шаблона класса Matrica
    Matrica<int > iM(inStr, inCol, 12);
    //Вывод на экран элементов матрицы
    iM.Chtenie(StringGrid, inStr, inCol);
} //Button1Click
```

Заменим теперь предпоследнюю строку кода функции такой строкой:

```
Matrica<String> iM(inStr, inCol, "Привет!");
```

В результате запуска приложения и нажатия командной кнопки в каждой ячейке таблицы строк будет выведено приветствие *Привет!* Далее между двумя последними строками новой редакции кода вставим строку:

```
iM.Zapisj (inStr, inCol, " Молодец!");
```

Теперь при нажатии командной кнопки в ячейках окажется похвала приложения *Молодец!* Как ясно из примеров, использование шаблона класса позволило очень компактно осуществлять генерацию объектов с заданным типом поля.

Для любых параметров шаблона могут быть заданы значения по умолчанию, например, в нашем приложении заголовок шаблона может быть таким:

```
template <class T= String> class Matrica
```

При использовании параметров шаблона по умолчанию список аргументов может оказаться пустым, однако при этом угловые скобки опускать *нельзя*:

```
Matrica<> iM(inStr, inCol, "Парметр по умолчанию");
```

У шаблона может быть несколько шаблонных типов. Если в нашем прежнем шаблоне добавить ещё один тип *T1*, то его описание может измениться, например, так:

```
template <class T, class T1> class Matrica
{
    T *Tx;//Содержимое ячеек матрицы
    int in_St;// Число строк
    int in_Cl;//Число столбцов
public:
    Matrica(int in_Str, int in_Col, T Txx, T1 T1xx);
    ~Matrica(){delete [] Tx;};
    void Zapisj (int in_Str, int in_Col, T Txx, T1 T1xx);
    void Chtenie (TStringGrid *StringGr, int in_Str, int in_Col);
};//Matrica
```

Заголовки реализаций шаблонных методов станут такими:

```
template <class T, class T1> Matrica<T, T1>::Matrica(int
                                in_Str, int in_Col, T Txx, T1 T1xx)
template <class T, class T1> void Matrica<T, T1>::Zapisj (int
                                in_Str, int in_Col , T Txx, T1 T1xx)
template <class T, class T1> void Matrica<T, T1>::Chtenie
                                (TStringGrid *StringGr, int in_Str, int in_Col)
```

Порождение экземпляра нового шаблонного класса, обновление данных и вывод их на экран:

```
Matrica<String, char> iM(inStr, inCol, "Привет!", 'Я');
iM.Zapisj (inStr, inCol, "Здравствуйте!", 'Ы');
iM.Chtenie(StringGrid, inStr, inCol);
```

Параметр типа *char* можно вывести, например, в *StringGrid[0][0]*.

В списке параметров шаблона *разрешается помещать* параметры с заданным типом, например:

```
template <class T, class T1, int Data= 25> class Matrica
```

Этот тип может быть произвольным, за исключением типа *void* и всех *веществен-*

ных типов. В нашем примере в качестве параметра *Data* использована константа. В общем же случае разрешается применять константное выражение, однако выражения, содержащие переменные, использовать в качестве фактических параметров шаблона *нельзя*.

Ниже перечислим правила описания шаблонов:

- ☐ Определение шаблона должно быть глобальным. Поэтому локальные классы не могут порождаться с использованием механизма шаблонов классов.
- ☐ Шаблоны методов не могут быть виртуальными.
- ☐ Шаблоны классов могут содержать статические элементы, дружественные функции.
- ☐ Шаблоны могут быть производными как от шаблонов, так и от обычных классов.

Остановимся теперь на достоинствах и недостатках шаблонов классов. Шаблоны представляют собой эффективный безопасный механизм обработки различных типов данных, который ещё можно назвать параметрическим полиморфизмом. Однако следует иметь в виду, что программа, включающая шаблоны, содержит полный код для каждого порождённого типа, что увеличивает размер исполняемого файла и несколько замедляет компиляцию. Кроме того, с некоторыми типами данных шаблоны могут работать не так эффективно, как с другими.

14.13. Отличие структур и объединений от классов

Структуры и объединения (см. тринадцатый урок) – это частные случаи классов. По умолчанию у структур базовый класс наследуется как общедоступный, методы и поля структур по умолчанию также общедоступны. У классов все эти элементы по умолчанию являются *закрытыми*. Структуры не имеют спецификатора доступа *published*, поэтому они не могут наследовать классы *VCL*, их элементы недоступны в *Инспекторе Объектов*.

Перечислим теперь отличия объединений от классов.

- ☐ В объединениях отсутствуют спецификаторы доступа, все их элементы наследуются как общедоступные.
- ☐ Объединение не может участвовать в иерархии классов.
- ☐ Элементами объединений не могут быть объекты, содержащие конструкторы и деструкторы, поскольку в разные моменты времени в их полях могут находиться данные различных типов.
- ☐ Объединение может иметь конструктор и нестатические методы.
- ☐ В анонимном объединении нельзя описывать (использовать) методы.
- ☐ Объединение не может содержать виртуальные методы и деструкторы.

Вопросы для самоконтроля всего урока

- ☐ Объясните сущность инкапсуляции и наследования.
- ☐ Назовите три разновидности полиморфизма.
- ☐ Какое правило лежит в основе совместимости объектных типов?
- ☐ В чём заключается раннее и позднее связывание? Укажите их достоинства и недостатки.
- ☐ Зачем применяются чисто виртуальные методы? Как называются классы, в которых определены чисто виртуальные методы?
- ☐ Какие функции называются полиморфными?
- ☐ В каком случае должны переопределяться виртуальные методы?
- ☐ Чем отличаются клиенты и дружественные функции класса?
- ☐ Где хранятся статические поля и константы класса?
- ☐ Перечислите отличительные черты прототипов конструктора и деструктора класса.
- ☐ С какой целью осуществляется перегрузка конструкторов?
- ☐ Для каких классов иерархии реализация конструктора может осуществляться вне описания класса?
- ☐ В чём заключается удобство применения деструкторов? В каких случаях деструкторы следует вызывать явно?
- ☐ Укажите недостатки побитового копирования объектов.
- ☐ Поля каких типов вынуждают разрабатывать конструкторы копирования?
- ☐ Что записывается в скрытое поле-указатель *this*? Каков механизм осуществления вызова метода для выбранного поля конкретного объекта?
- ☐ Перечислите операции, необходимые для программного порождения визуальных объектов.
- ☐ Где должны описываться методы локальных классов?
- ☐ Следует ли при генерации объекта по шаблону класса явно указывать тип шаблонных параметров?

14.14. Задача для программирования

Задача 14.1. Для закрепления навыков работы с классами разработайте приложение, в котором объявлен класс, включающий только два закрытых поля. Первое поле целочисленных данных инициализируется случайным числом при помощи конструктора класса. Второе поле является указателем на массив объектов, тип которых тождественен этому же классу. Помимо конструктора и деструктора разработайте открытые методы класса для вывода в таблицу строк целочисленных полей массива объектов (второго поля класса) и метод сортировки по

возрастанию упомянутых значений. Объявите глобальный объект *Объект* указанного класса. Таблица строк с двумя строками должна порождаться динамически при нажатии командной кнопки. В первую её строку при помощи метода вывода запишите содержимое целочисленных полей из массива объектов второго поля объекта *Объект* до вызова метода сортировки, а во вторую строку – после вызова метода сортировки. Интерфейс приложения может иметь вид, показанный на рис. 14.2.

14.15. Вариант решения задачи

```
...
const int in= 5;//Длина массива

class TKlass_Chislo
{
    int Chislo;
    TKlass_Chislo *Mas_Klas[in];
public:
    TKlass_Chislo(void);//Конструктор
    //Для инициализации поля Mas_Klas
    void Inic_Massiva(void);
    int Chtenie(void);//Для чтения поля Chislo
    void Sortirovka(void);
    void Vubod(TStringGrid *StringGr, int Stroka);
    ~TKlass_Chislo(void);//Деструктор
};//TKlass_Chislo

TKlass_Chislo::TKlass_Chislo(void)
{
    //Случайное число в интервале 0-(in*in-1)
    Chislo= random(in*in);
}

//TChislo_Klass::TChislo_Klass
```

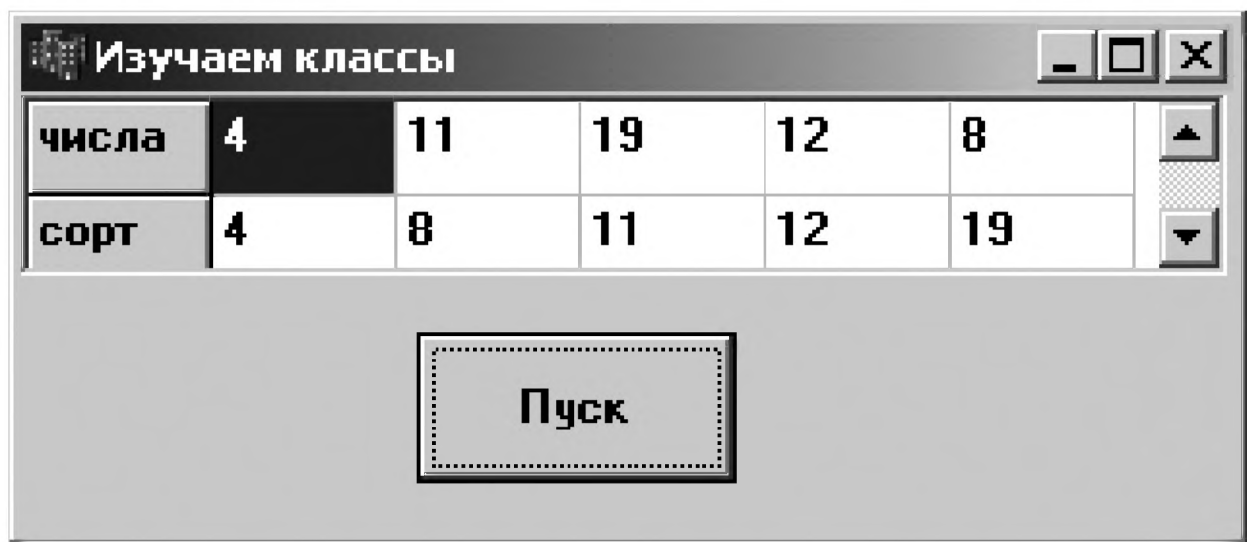


Рис. 14.2. Интерфейс приложения *Изучаем классы*

```

//Метод инициализации массива указателей
//на класс TKlass_Chislo
void TKlass_Chislo::Inic_Massiva(void)
{
    for (int ij= 0; ij< in; ij++)
        Mas_Klas[ij]= new TKlass_Chislo;
} //TKlass_Chislo::TChislo_Klass

//Сортировка методом пузырька
void TKlass_Chislo::Sortirovka(void)
{
    bool Sort;
    do
    {
        Sort= false;
        for (int ij= 0, iByfer; ij< in-1; ij++)
            if (Mas_Klas[ij]->Chislo>Mas_Klas[ij+1]->Chislo)
            {
                iByfer= Mas_Klas[ij]->Chislo;
                Mas_Klas[ij]->Chislo= Mas_Klas[ij+1]->Chislo;
                Mas_Klas[ij+1]->Chislo= iByfer;
                Sort= true;
            } //if
    } //do
    while(Sort);
} //TKlass_Chislo::TChislo_Klass

int TKlass_Chislo::Chtenie(void)
{
    return Chislo;
} //TKlass_Chislo::Chtenie

void TKlass_Chislo::Vubod(TStringGrid *StringGr, int Stroka)
{
    for (int ij= 0; ij< in; ij++)
        StringGr->Cells[ij+1][Stroka]= Mas_Klas[ij]->Chtenie();
} //TKlass_Chislo::Vubod

TKlass_Chislo::~TKlass_Chislo(void)
{
    delete [] Mas_Klas;
} //TKlass_Chislo::~TKlass_Chislo

TKlass_Chislo Objekt; //Порождён глобальный объект

void __fastcall TForm1::PyskClick(TObject *Sender)
{
    TStringGrid *StringGrid= new TStringGrid(this);
    StringGrid->Parent= this;
    StringGrid->RowCount= 2;
    StringGrid->ColCount= in + 1;
    StringGrid->FixedRows= 0;
    StringGrid->FixedCols= 1;

```

```
StringGrid->DefaultColWidth= 48;//Ширина столбца
//Высота окна
StringGrid->Height= StringGrid->DefaultRowHeight*2;
StringGrid->Cells[0][0]= "числа";//"Случайные числа";
StringGrid->Cells[0][1]= "сорт";//"Упорядоченные значения";
Objekt.Inic_Massiva();
Objekt.Vubod(StringGrid, 0);//Исходные значения
Objekt.Sortirovka();
Objekt.Vubod(StringGrid, 1);//Числа после сортировки
}//PyskClick
```



Урок 15. Графика и мультипликация

15.1. Основные определения

Графика широко применяется в научных и инженерных исследованиях, поскольку графические иллюстрации *существенно* облегчают анализ результатов расчётов. В *C++Builder* для графических построений предусмотрены специальные компоненты, некоторые из них изучены на пятом уроке. Не взирая на их мощь, в конкретных условиях может оказаться эффективнее использовать *свои* графические разработки. Поэтому познакомимся с тем, как «делается» графика.

Однако прежде необходимо отметить, что графический режим многих языков программирования высокого уровня уже давно взят на вооружение для разработки различных пакетов прикладных программ систем автоматического проектирования (САПР), являющихся мощным инструментом исследовательских и опытно-конструкторских работ.

Судите сами. Ведь такие программы в обход кульманов, ватмана и карандашей позволяют *оперативно* выполнять проектирование механизмов, машин и конструкций, проводить всесторонние испытания и доводки (ещё *до* воплощения изделий в металле, камне и др. материалах), наглядно представлять их вид и работу при всех задаваемых разработчиком условиях. Поскольку при этом отсутствуют какие-либо затраты на производство и натурные испытания, то такой инструментарий не только колоссально сокращает время проектирования, но ещё и экономит средства, поэтому является действенным катализатором научно-технического прогресса.

Следует признать, за последние десять – пятнадцать лет *уже создано* достаточно *много* специальных высококачественных программ такого типа. Поэтому, господа, не изобретайте велосипед: многие типовые задачи решайте с их помощью.

Имеется, например, замечательная программа *SigmaPlot*. Она предназначена для построения графиков и анализа массивов данных. Мощный пакет программ *StatGraphics* хорошо справляется со статистической обработкой данных, табличным и графическим представлением ее результатов.

Существуют ещё пакеты программ визуального программирования, они также предназначены для автоматизации научно-исследовательских работ: *Математика*, *MathCad*, *MathLab* и др. Используя подобные пакеты, можно производить расчёты по очень широкому кругу математических вопросов и одновременно получать наглядное представление их решений без знаний какого-либо языка программирования высокого уровня.

Такие программы длительное время разрабатывали большие коллективы очень квалифицированных специалистов. Вследствие этого вам, дорогие читатели, создать что-либо подобное на первых порах будет трудновато. При этом мы нисколько не сомневаемся в вашем таланте и трудолюбии. Что поделаешь, если упомянутые коллективы начали творить несколько раньше вас.

По указанным причинам предлагаем вначале ограничиваться менее масштабными проектами. Вместе с тем отметим, что в инженерных задачах, решаемых при помощи *C++Builder*, всё же *очень часто* приходится привлекать графику. Поэтому рассмотрим её достаточно подробно, после создания неподвижных изображений перейдём к разработке видеороликов.

Но прежде разберёмся, что же представляет собой изображение на экране монитора? Изображение или картинка это набор мельчайших фрагментов – точек определенного цвета и яркости. Эти точки называются пикселями. По-английски фрагмент изображения – это *PICTure'S ELe ment*. Вместе буквы *C* и *S* произносятся аналогично букве *X*, поэтому в сокращении такой элемент получил название *pixel*. При помощи специальных функций можно управлять не только цветом и яркостью произвольной точки экрана, но и выводить геометрические примитивы (отрезки прямых, прямоугольники, эллипсы, секторы и др.), управлять цветом переднего и заднего плана, формировать палитру цветов и многое другое.

Как в кино и в телевидении, в дисплеях эксплуатируется инерционность человеческого зрения, проявляющаяся в том, что картинки, демонстрируемые чаще, чем 25 раз в секунду, не мелькают, а кажутся непрерывным изображением. Таким образом, картинку необходимо повторять минимум с частотой 25–30 Герц. При меньших частотах наблюдается вредное и утомительное для глаз подёргивание изображения.

Но чтобы изображение повторять, его прежде нужно запомнить. Так вот, для запоминания информации о каждом пикселе экрана (его цвете, яркости, местоположении) предназначена специальная память. Она называется *видеопамятью* или видеобуфером. Videобуфер относится к оперативной памяти, поскольку его информация исчезает с выключением компьютера. Но расположена эта память совершенно в другом месте – в видеоадаптере.

Видеоадаптер не только запоминает изображение, но ещё и повторяет его с заданной частотой. Он искусно обманывает наш мозг, используя инерционный недостаток зрения человека (который оказывается очень полезным). Видеоадаптер или просто адаптер представляет собой электронную плату (по-английски – *card*), состоящую из видеопамяти и контролера монитора. Адаптер по выполняемым функциям подобен географической карте, хранящей цветные точки в разных позициях листа. Поэтому его часто называют видеокартой. Может быть это название обусловлено неудачным переводом термина *videocard*, ведь по-английски *card* переводится, как карта.

С каждым годом повышаются возможности видеоадаптеров, их частота повторения картинок увеличивается. В настоящем она чаще всего не превышает 100 Герц. Повторяем, чем больше эта частота, тем менее заметно мерцание экрана

и меньше утомляемость глаз. Объём видеопамати также неуклонно растёт, у современных адаптеров он уже несколько десятков мегабайт.

С графикой немного разобрались. Перейдём теперь к мультипликации, заявленной в названии урока. Движение фрагментов изображения можно осуществлять четырьмя способами:

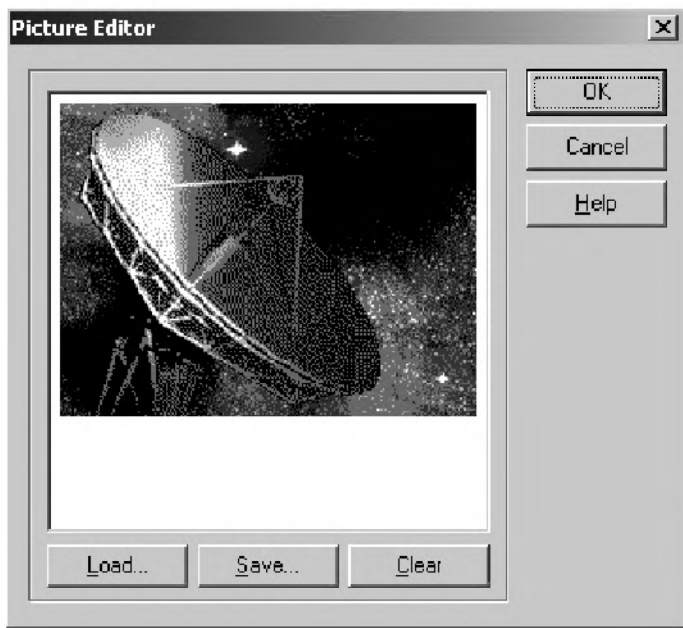
1. Рисовать изображение различными цветами переднего плана (пера), а затем, спустя время, стирать его – т.е. выводить то же самое изображение цветом фона. Затем вновь размещать это же изображение в несколько смещенном положении и стирать его после задержки на экране. Траектория движения определяется программой.
2. Выводить изображение прежним способом, но стирать его различными фигурами: прямоугольником, эллипсом, сектором и др. Цвет контура и заливки таких фигур совпадает с цветом фона, габариты фигур несколько превышают габариты изображения.
3. Изображение выводить также как и в предшествующих способах, однако его стирание обеспечивается применением специального режима пера *mpNotXor*. При установке такого режима *повторный* вывод изображения на прежнем месте заменяет его цветом заднего плана участка экрана, на который выводилось изображение первый раз.
4. Изображение подвижного фрагмента записать в буфер, который размещается в динамической памяти. Затем это изображение выводить из буфера последовательно в разные места экрана. При этом, перед очередным выводом, изображение в предшествующем месте (после задержки) стирать.

Все перечисленные варианты воплощения мультипликации детально иллюстрируются и разъясняются в настоящем уроке. Поэтому не волнуйтесь, если что-то (или всё) в этих вариантах пока не ясно. Сейчас же перейдём, может быть, к самому важному для вас вопросу.

15.2. Логотип приложения

В вашей будущей процветающей фирме по разработке приложений, конечно же, будет оригинальный логотип, появляющийся в ходе запуска *всех* выпущенных ею программных продуктов! Как при помощи таймера и немодальной формы вывести на 5 секунд приветствие рассказывалось на одиннадцатом уроке (см. задачу 11.2). Теперь вместо текстового приветствия покажем картинку-логотип. Её можно нарисовать самостоятельно, используя какой-либо графический редактор, отсканировать фотографию или иллюстрацию, взять готовое изображение, например, из папки *C:\Program Files\Common Files\Borland Shared\Images\Splash\16Color*. В нашем учебном приложении используется картинка *technlgy.bmp*. Для её вывода на интерфейс применим компонент *Image* (изображение, картинка), он расположен на вкладке *Additional Палитры Компонентов* и выглядит вот так: . Этот компонент является контейнером графических изображений.

Познакомимся с его основными возможностями в ходе разработки приложения с заголовком *Картинка*. Для этого следуйте нашим рекомендациям. Открой-

Рис. 15.1. Окно *Picture Editor*

те новое приложение с упомянутым заголовком, породите на нём объект с именем по умолчанию *Image1*, установите габариты объекта близкими габаритам формы. Далее, при выделенном объекте *Image1*, перейдите в *Инспектор Объектов* и сделайте активным его свойство *Picture* (картина). Это свойство позволяет хранить картинки и показывать их на форме.

Теперь, нажмите на кнопку с тремя точками, расположенную справа от свойства *Picture*, либо сделайте двойной щелчок в поле ввода этого свойства. В любом из этих вариантов перед вами откроется окно *Picture Editor* (см. рис. 15.1), позволяющее загрузить в объект *Image1* какой-либо графический файл (кнопкой *Load...*), а также сохранить открытый файл под новым именем или (и) в новом каталоге (кнопкой *Save....*). Последовательно выполним упомянутые операции.

Щёлкните по кнопке *Load* для загрузки требуемого файла. После щелчка откроется окно, показанное на рис. 15.2. В ходе перемещения курсора в каталоге графических файлов (например, в указанном выше) в правом окне отображается изображение, хранящееся в выбранном файле; над изображением (в круглых скобках) приводятся его горизонтальный и вертикальный габариты в пикселях.

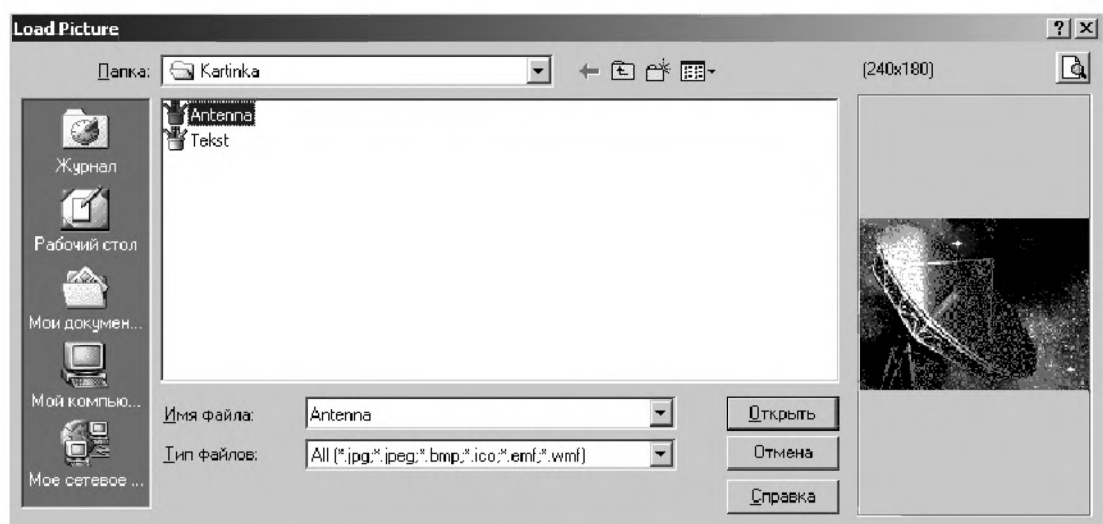


Рис. 15.2. Окно, предназначенное для загрузки графического файла

Остановите свой выбор на файле *technlgy.bmp* и загрузите его, нажав кнопку *Открыть*. Далее нажмите кнопку *ОК*.

При нажатии кнопки *Save...* в окне *Picture Editor* (рис. 15.1), увидите окно, аналогичное окну на рис. 15.2, в нём лишь вместо кнопки *Открыть* находится кнопка *Сохранить*. Сохраните этот файл, например, под именем *Antenna.bmp* в каталоге с разрабатываемым приложением (для удобства последующей работы с ним). Теперь приложение украшено картинкой. Эта картинка сохраняется в самом приложении, что даёт возможность поставлять его (продавать) без упомянутого графического файла.

Познакомимся с несколькими свойствами объекта *Image1* (объектов типа *TImage*).

- ☐ *Center* (при константе *true*) центрирует изображение в прямоугольнике объекта *Image1* (если габариты изображения меньше габаритов *Image1*).
- ☐ *AutoSize* (при *true*) габариты объекта *Image1* уравниваются с габаритами помещаемой в него картинки. При *false* изображение может выходить за границы объекта, либо габариты картинки окажутся меньше габаритов объекта.
- ☐ *Stretch* (растягивание, удлинение, растяжение) позволяет (при *true*) растягивать габариты рисунка до габаритов объекта. Во многих случаях такое растягивание по горизонтали и вертикали оказывается различным, что приводит к искажению картинки. Поэтому это свойство разумно активизировать только, когда картинка представляет собой какой-то узор.
- ☐ *Transparent* (прозрачность) используется при наличии перекрывающихся картинок. Если для верхнего изображения в этом свойстве установить константу *true*, то будет видна картинка, расположенная за ним. Часто такую возможность свойства используют при наложении на картинку надписей, выполненных в виде битовых матриц. Применить это свойство для других (кроме *.BMP*) типов файлов нельзя. С различными типами графических файлов познакомимся в п. 15.4.

Используем свойство *Transparent* для наложения на прежнюю картинку надписи, выполненной в виде битовой матрицы. Надпись изготовим при помощи встроенного в *C++Builder* графического редактора *Image Editor* (он использовался в п. 9.5.3).

Изготовление надписи с прозрачным фоном

1. Запустите редактор *Image Editor*. *Tools/Image Editor/File/New.../Bitmap File (.bmp)*.
2. Установите следующие его параметры: *Width* – 300, *Height* – 100; активизируйте режим *VGA (16 colors)*.
3. При помощи инструмента *Text* напишите несколько строк текста (см. рис. 15.3), запомните этот рисунок в файле с именем *Text.bmp* и расположите его в каталоге приложения (лишь для удобства последующей работы с ним).

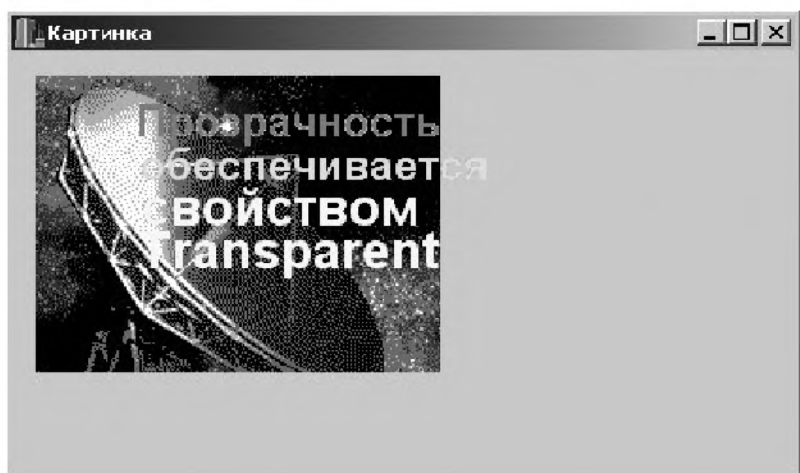



Рис. 15.3. Иллюстрация применения свойства *Transparent*

4. На интерфейсе приложения *Картинка* (сделайте его копию и работайте с ней) дополнительно расположите объект *Image2*, и прежним способом загрузите в него файл *Text.bmp*. Частично наложите объект *Image2* на объект *Image1*: окажется, что объект *Image2* экранирует перекрытую им часть объекта *Image1*.
5. После активизации свойства *Transparent* объекта *Image2* (выбором константы *true*) его задний план окажется прозрачным (см. рис.15.3).

15.3. Приложение для просмотра графических файлов

На основе приложения *Картинка* разработаем программу для просмотра *любых* графических файлов. Для этого выполните следующую последовательность действий.

1. Сделайте копию приложения *Картинка* и переименуйте ее в *Просмотр графических файлов*.
2. Породите на форме объект *OpenPictureDialog1*, его компонент *OpenPictureDialog*  расположен на странице *Dialogs Палитры компонентов*. Этот объект вызывает диалоговое окно поиска, открытия и предварительного просмотра файла-изображения (см. рис. 15.2).
3. Удалите объект *Image2* и породите объект-меню с единственным пунктом *Файл*. Как создаётся меню, рассматривалось в п. 9.5.1. В функцию по обработке нажатия этого пункта код будет вписан позже.
4. В файле реализации укажите директиву компилятору:

```
#include <jpeg.hpp>
```

Если эту директиву не использовать, то объект *Image1* будет отображать *только* содержимое файлов с расширениями *bmp*, *ico* и *wmf*, при попытке загрузить файл с расширением *jpg* в методе *LoadFromFile* возникнет исключительная ситуация *EInvalidGraphic*. Применение директивы позволяет осуществить показ изображений, хранящихся в файлах и с расширением *jpg*.

5. В функцию по обработке нажатия пункта меню впишите оператор условия:

```
if (OpenPictureDialog1->Execute())
    Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
```

Этот оператор загружает в свойство *Picture* объекта *Image1* файл, выбранный пользователем в диалоговом окне.

6. Габариты объекта *Image1* сделайте близкими габаритам формы.

Поскольку загружаемые изображения обычно имеют *разные* габариты, то в одних случаях картинка может не помещаться *полностью* в окне объекта *Image1* (следовательно, и на форме), а в других – окажется значительно меньше этого окна, (при этом центр окна и картинки, как правило, не совпадают). Поэтому сделаем так, чтобы габариты формы настраивались на размеры изображения, которое *автоматически* центрируется в окне формы.

С этой целью в приведенном ниже фрагменте кода размеры клиентской области формы (по высоте и ширине) на 10 пикселей превышают габариты объекта *Image1*, который располагается *посредине* формы. При этом габариты объекта *Image1* при помощи активизации свойства *AutoSize* автоматически адаптируются к размеру картинки. Последняя операция выполняется первой строкой кода тела оператора *if*. Если эту операцию осуществить в *Инспекторе Объектов*, то указанную строку в коде можно исключить. Теперь внимательно рассмотрите программную реализацию описанных выше действий.

```
if (OpenPictureDialog1->Execute())
{
    Image1->AutoSize= true;
    Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
    Form1->ClientHeight= Image1->Height + 10;
    Image1->Top= Form1->ClientRect.Top +
                (Form1->ClientHeight - Image1->Height)/2;
    Form1->ClientWidth= Image1->Width + 10;
    Image1->Left= Form1->ClientRect.Left +
                (Form1->ClientWidth - Image1->Width)/2;
} //if
```

Здесь *ClientRect.Top* и *ClientRect.Left* определяют положение верхнего левого угла прямоугольника клиентской области формы. Интерфейс программы показан на рис. 15.4.

15.4. Типы графических файлов

Для хранения изображений в *C++Builder* используются такие типы графических файлов: битовые матрицы, пиктограммы и метафайлы. В указанных типах файлов применяются разные способы хранения изображений и доступа к ним. *Битовая матрица* (файл с расширением *.bmp*) – это набор пикселей заданного цвета и интенсивности. В разных дисплеях число пикселей в строке и число строк может не совпадать. Однако способ хранения этой информации позволяет отобразить картинку (без заметных искажений) практически на *любом* дисплее и не зависит от конфигурации используемого компьютера.



Рис. 15.4. Интерфейс приложения *Просмотр графических файлов*

Пиктограммы (файлы с расширением *.ico*) представляют собой битовые матрицы уменьшенных размеров, применялись ранее (см., например, п. 9.5.3) в качестве значков приложений, для оформления кнопок быстрого доступа. Одно из основных различий между пиктограммами и битовыми матрицами состоит в том, что пиктограммы *невозможно* масштабировать: они всегда сохраняют свой стандартный размер.

Битовые матрицы и пиктограммы относятся к *растровой* графике, поскольку их изображения хранятся в виде набора цветных точек (пикселей). Для хранения таких изображений (за исключением пиктограмм) расходуется большой объём дисковой памяти, при значительном увеличении размеров картинки её качество существенно ухудшается (например, на наклонных линиях (границах) проявляются ступеньки). Редакторами растровой графики являются, например, *MS-Paint* и *Image Editor*.

Метафайлы (Metafiles) относятся к векторной графике. В редакторах векторной графики (например, *MS-Word*, *CorelDRAW*) изображение формируется набором примитивов (отрезков кривых и прямых, плоских и объёмных геометрических фигур), в файле хранятся *только* операторы, описывающие их математические выражения (формулы и наборы коэффициентов). Это позволяет сократить (в десятки раз) объём требуемой памяти, масштабирование такого изображения осуществляется без ухудшения качества. В табл. 15.1 перечислены все расширения файлов, с которыми может работать *C++Builder*.

Таблица 15.1. Типы файлов, которые могут обрабатываться C++Builder

Тип файла	Расширение
JPEG Image File	<i>jpg, jpeg</i>
Битовые матрицы (<i>Bitmaps</i>)	<i>bmp</i>
Пиктограммы	<i>ico</i>
Enhanced Metafiles	<i>emf</i>
Metafiles	<i>wmf</i>

Файлы с расширениями *jpg, jpeg* представляют собой битовые матрицы, хранящиеся в запакованном формате. Поэтому архивация таких файлов практически не уменьшает их объём, а объём этих файлов в десятки раз меньше, чем тех же картинок в формате *bmp*.

15.5. Объекты для хранения изображений, открытие, сохранение и переименование файлов

В C++Builder для запоминания изображений в оперативной памяти имеется класс *TPicture*, объекты этого класса (типа) могут хранить в себе и битовые матрицы, и пиктограммы, и метафайлы. Это оказывается возможным, поскольку класс *TPicture* является производным от классов *TBitmap*, *TIcon* и *TMetafile*: в нём осуществляется множественное *открытое* наследование указанных классов.

В объектах классов *TBitmap*, *TIcon* и *TMetafile* можно хранить *только* соответствующий их названию тип файла. Если в разрабатываемом приложении тип графического файла чётко определён, то разумнее порождать объект *соответствующего* типа, поскольку это несколько ускоряет обработку изображений: устраняется лишняя цепочка доступа к нужному свойству. Однако в случае применения такого прямого механизма доступа следует контролировать тип обрабатываемого файла. При попытке обработать «чужой» файл генерируется исключительная ситуация.

Во всех *четырёх* указанных выше классах имеется целый набор полезных переопределяемых методов. Методы загрузки *LoadFromFile* и сохранения в файл *SaveToFile* знакомы нам по классам *TMemo*, *TRichEdit*, *TStringList*, применяемым на девятом уроке. Для *присваивания* во всех четырёх классах служит метод *Assign*, впервые использованный в п.11.6.6 для загрузки (присваивания) выбранного шрифта. Эти и многие другие методы перечисленных классов изначально (первый раз) описаны в их общем абстрактном базовом классе *TGraphic*.

Познакомимся с техникой обработки графических файлов на примере следующего учебного приложения. В оперативной памяти (в объекте *Bitmap*) запоминается изображение битовой матрицы. Затем это изображение переписывается в находящийся на форме объект *Image1* (его содержимое сразу же проявляется на форме). Далее в изображение объекта *Image1* вносятся изменения. Исходное и

модернизированное изображения запоминаются в *отдельных* файлах, а затем в объекте *Image1* модернизированное изображение восстанавливается путём записи в него исходного изображения из объекта типа *Bitmap*.

Помимо знакомства с обработкой графических файлов в этом разделе продолжим изучение диалогов (см. п. 11.6.6). Напоминаем, заготовки всех диалогов находятся на закладке *Dialogs Палитры компонентов*. Ниже приведен код, реализующий упомянутую выше задачу.

```
...
//Объявлен и инициализирован глобальный указатель BitMap на
//графический объект, Graphics— имя модуля
Graphics::TBitmap *BitMap= new Graphics::TBitmap();
String MyFName= ""; //Глобальное имя файла

void __fastcall TForm1::FileClick(TObject *Sender)
{
    //Загрузка изображения в BitMap и Image1
    if (OpenPictureDialog1->Execute())
    {
        BitMap->LoadFromFile(OpenPictureDialog1->FileName);
        Image1->AutoSize= true;
        Image1->Picture->Assign(BitMap);
        Form1->ClientHeight= Image1->Height + 10;
        Image1->Top= Form1->ClientRect.Top +
            (Form1->ClientHeight - Image1->Height)/2;
        Form1->ClientWidth= Image1->Width + 10;
        Image1->Left= Form1->ClientRect.Left +
            (Form1->ClientWidth - Image1->Width)/2;
    } //if
} //FileClick

void __fastcall TForm1::SaveClick(TObject *Sender)
{
    if (MyFName != "") //Если имя не пустая строка (известно),
    //то нет необходимости обращаться к диалогу
        Image1->Picture->SaveToFile(MyFName);
    //Если имя файла неизвестно, то он сохраняется с помощью диалога
    else
        if (SaveDialog1->Execute())
        {
            MyFName= SaveDialog1->FileName;
            Image1->Picture->SaveToFile(SaveDialog1->FileName);
        } //if
} //SaveClick

void __fastcall TForm1::SpoilClick(TObject *Sender)
{
    //Вывод горизонтального отрезка красного цвета
    int Nachalo= (Form1->Image1->Width)/10,
        Konec= Form1->Image1->Width,
        Sredina= (Form1->Image1->Height)/2;
    for (int ii= Nachalo; ii< Konec; ii++)
        Form1->Image1->Canvas->Pixels[ii][Sredina]= clRed;
} //SpoilClick

void __fastcall TForm1::RestoreClick(TObject *Sender)
```

```

{
    //Восстанавливает изображения в Image1 из BitMap
    Image1->Picture->Assign(BitMap);
} //RestoreClick

void __fastcall TForm1::Save_AsClick(TObject *Sender)
{
    //Присваивается имя существующего файла
    SaveDialog1->FileName= MyFName;
    //Сохранение с новым именем
    if (SaveDialog1->Execute())
    {
        MyFName= SaveDialog1->FileName;
        Image1->Picture->SaveToFile(SaveDialog1->FileName);
    } //if
} //Save_AsClick

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    delete BitMap;
    BitMap=NULL;
} //FormDestroy

```

Интерфейс приложения показан на рис. 15.5. Функции *FileClick*, *SaveClick*, и *Save_AsClick* обеспечивают открытие, сохранение и копирование файла. Каждая из них обрабатывает нажатие соответствующего пункта меню на интерфейсе программы. Функция *FormDestroy* обрабатывает событие *OnDestroy* формы, связанное с её уничтожением.

В функции *FileClick* при помощи стандартного окна открытия файла и метода *LoadFromFile* в объект *BitMap* загружается изображение из выбранного пользователем файла. После активизации свойства *AutoSize* объекта *Image1*, в этот объект при помощи свойства *Picture* и метода *Assign* записывается (присваивается) изображение из объекта *BitMap*. Таким образом, осуществляется перезапись изображения из одного объекта в другой. Это сделано для того, чтобы изображение в объекте *Image1* (видимое на интерфейсе) после модернизации можно было бы восстановить при помощи исходного изображения, хранящегося в объекте *BitMap*.

Далее в функции *FileClick* осуществляется подбор габаритов клиентской области формы к габаритам загруженной картинки и центрирование последней на форме. Эти действия реализованы точно так же, как в приложении *Просмотр графических файлов* (строки кода были скопированы из упомянутого приложения).

В функции *SaveClick* осуществляется сохранение изображения из объекта *Image1* в файл, имя и каталог размещения которого указал пользователь в стандартном окне сохранения файла. Для сохранения используется свойство *Picture* и его метод сохранения *SaveToFile*. Работа диалога выполняется по следующему сценарию. Если имя сохраняемого файла *MyFName* уже известно (не равно пустой строке, поскольку пользователь его уже выбрал или набрал в предшествующем случае обращения к диалогу), то нет необходимости обращаться к диалогу: изображение сохраняется методом *SaveToFile*. Если же имя файла неизвестно

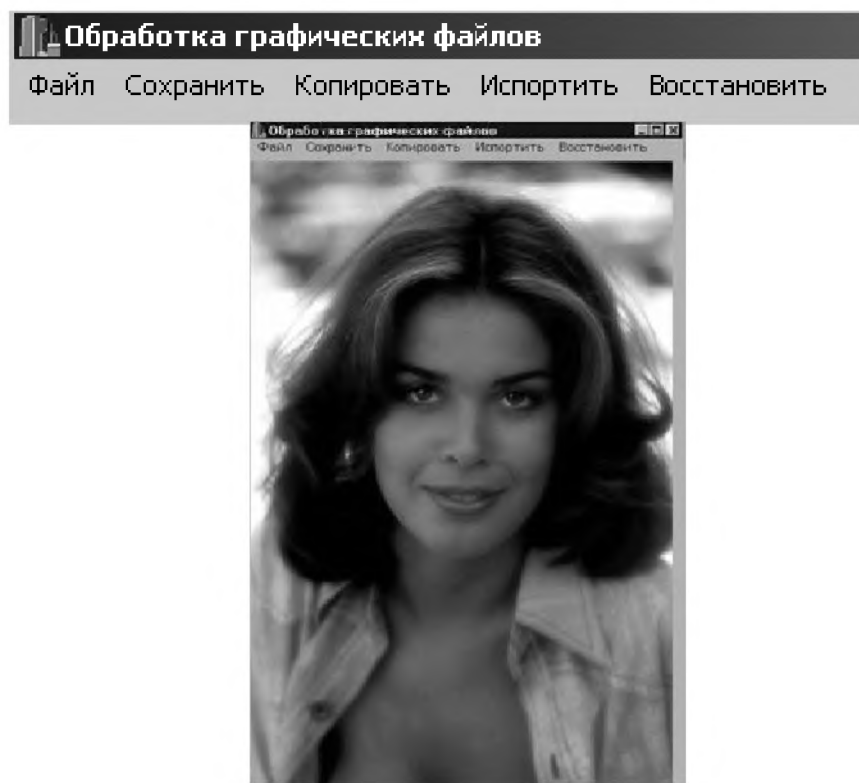


Рис. 15.5. Интерфейс приложения *Обработка графических файлов* (нижняя панель) и увеличенный заголовок и меню программы (верхняя панель)

(пустая строка), то изображение сохраняется с помощью диалога *SaveDialog1*: пользователь выбирает или набирает имя файла, определяет каталог и после нажатия кнопки *Открыть* происходит сохранение изображения в файле с выбранным именем в указанном каталоге.

Функция *SpoilClick* обеспечивает модернизацию изображения, хранимого в объекте *Image1*. В нашем упрощённом учебном приложении эта модернизация заключается лишь в том, что картинка перечёркивается посередине прямым горизонтальным отрезком красного цвета. Фактически изображение портится, поэтому и пункт меню, вызывающий эту функцию (см. рис. 15.5), имеет заголовок *Испортить*, а имя *Spoil*.

В этой функции построение отрезка прямой осуществляется выводом в цикле *for* самого простого примитива – точки. Этот процесс осуществляется так: задаются ордината горизонтального отрезка (переменная *Sredina*) и абсциссы его начала и конца (переменные *Nachalo* и *Kones*); далее в цикле *for* последовательно по горизонтали выводятся пиксели красного цвета между абсциссами *Nachalo* и *Kones*. Пиксель (и все другие примитивы) можно вывести при помощи свойства *Canvas* (холст для рисования) объекта *Image1*. Этим свойством обладают объекты и многих других типов, например: *TForm*, *TBitMap*, *TPaintBox*.

Свойства визуальных объектов, порождённых на форме, отображаются в *Инспекторе Объектов*. Упомянутые объекты являются экземплярами классов, часто имеющих предков. Так вот родительские классы (или одноимённые свойства визуальных объектов) *не отражаются* в *Инспекторе Объектов*, к ним возможен *только программный* доступ. В рассмотренном выше примере недоступно в *Инспекторе Объектов* свойство *Canvas* (далее также – канва). В этом уроке будут

использоваться также *вложенные* свойства канвы, они инкапсулированы уже в класс (свойство) *Canvas*: наследуются этим классом (как и свойство *Canvas* и недоступны в *Инспекторе Объектов*).

Положение пикселя на поверхности холста характеризуется горизонтальной (*X*) и вертикальной (*Y*) координатами. Координата *X*, как принято в математике, возрастает слева направо. Однако координата канвы *Y* увеличивается сверху вниз. Поэтому, например, левый верхний угол формы (прямоугольника её клиентской области) имеет координаты (0, 0), правый нижний – (*ClientWidth*, *ClientHeight*).

У свойства *Canvas* имеется много инструментов-методов, предназначенных для вывода геометрических фигур (с основными методами познакомимся в следующем разделе). Для зажигания на канве точки красного цвета (*clRed*) и заданными координатами по горизонтали (*ii*) и вертикали (*Sredina*) используется метод *Pixels*:

```
Form1->Image1->Canvas->Pixels[ii][Sredina]= clRed;
```

Метод *Pixels* – это двумерный массив, в ячейках которого хранится информация о цветах пикселей канвы (с заданными координатами, местоположениями).

Для восстановления испорченного изображения вызывается функция *RestoreClick*, предназначенная для обработки нажатия пункта горизонтального меню *Восстановить*. В этой функции процесс восстановления осуществляется при помощи присваивания свойству *Picture* объекта *Image1* методом *Assign* исходного изображения, хранимого в объекте *BitMap*: *Image1->Picture->Assign(BitMap)*;

Для создания файла-копии изображения с новым именем применяется функция *Save_AsClick*, она вызывается при нажатии пункта горизонтального меню *Копировать*. Конечно, без привлечения этой функции в приложении вполне можно обойтись. Однако её использование иллюстрирует применение диалога *SaveDialog1* для создания *копии* файла. В первой строке этой функции реализуется предложение пользователю выбрать имя по умолчанию для нового имени файла, это имя осталось в идентификаторе *MyFName* глобальной переменной при предшествующем вызове этого же диалога:

```
SaveDialog1->FileName= MyFName;
```

Следующий оператор (*if*) открывает диалог, после выбора имени файла и его местоположения в файловой системе компьютера, создаётся копия файла.

15.6. Графические примитивы и инструменты для их построения

15.6.1. Карандаш и кисть

Графические примитивы – это методы класса *TCanvas*, которые определяют *только вид* и *параметры* геометрической фигуры. За её прорисовку и заливку внутренней области отвечают свойства канвы *Pen* (карандаш) и *Brush* (кисть): класс *TCanvas* осуществляет множественное наследование различных классов, среди которых имеются и классы *TPen* и *TBrush*. Карандаш и кисть являются теми

инструментами, которыми необходимо воспользоваться для задания толщины и цвета линии примитива, цвета и способа закраски его внутренней области. Упомянутые классы инкапсулируют другие классы, к полям и методам объектов этих классов доступ осуществляется при помощи операции стрелка « \rightarrow ». Напоминаем, что такая операция применяется в связи с тем, что все визуальные и не визуальные компоненты *VCL* являются указателями на соответствующие классы. Класс *TCanvas* в свою очередь включён в классы *TForm*, *TImage*, *TBitmap* и др.

Свойство *Pen* при помощи своих подсвойств *Color* и *Width* определяет цвет и толщину линии (в пикселях). Подсвойство *Style* (вид) указывает вид линии:

- ☐ *psSolid* – сплошная;
- ☐ *psDash* – штриховая;
- ☐ *psDot* – точечная;
- ☐ *psDashDot* – пунктирная;
- ☐ *psDashDotDot* – штрих с двумя точками;
- ☐ *psClear* – границы примитива не отображаются.

В свойстве *Brush* подсвойство *Color* определяет цвет заливки замкнутой области, а подсвойство *Style* задает стиль (или вид) штриховки области. Ниже приведены допустимые для свойства *Style* значения и соответствующие им шаблоны штриховки:

- ☐ *bsSolid* – сплошная заливка цветом кисти;
- ☐ *bsHorizontal* – горизонтальными прямыми;
- ☐ *bsVertical* – вертикальными прямыми;
- ☐ *bsFDDiagonal* – прямыми вдоль главной диагонали;
- ☐ *bsDDiagonal* – прямыми вдоль побочной диагонали;
- ☐ *bsCross* – вертикальными и горизонтальными прямыми;
- ☐ *bsDiagCross* – прямыми вдоль главной и побочной диагоналей.

Команды графических построений очень часто помещают в функцию обработки событий *OnPaint* для объектов типа *TForm*, *TImage* и др. Эта функция перерисовывает изображение всякий раз, когда оно портится выводом какого-либо окна *Windows*. Упомянутое событие возникает и в момент запуска приложения, когда его интерфейс появляется на экране первый раз.

В качестве иллюстрации возможностей упомянутых инструментов запустите приложение *Флаг*, которое выводит три прямоугольника, два из них образуют флаг Украины, третий – показывает один пример вида линии и типа штриховки (см. рис. 15.6). Для вычерчивания прямоугольника используется метод *Rectangle*, первые его два параметра определяют координаты верхнего левого угла, а последние – координаты нижнего правого угла. Следующий код разместите в функции *FormPaint* (обработка события событий *OnPaint*).

```
//Флаг Украины
Canvas->Pen->Style= psClear;

//Верхняя часть флага
Canvas->Brush->Color= clBlue;//Синий цвет заливки
Canvas->Rectangle(30,30,200,90);
```



Рис. 15.6. Иллюстрация работы свойств *Pen* и *Brush*

```
//Нижняя часть флага
Canvas->Brush->Color= clYellow;//Жёлтый цвет заливки
Canvas->Rectangle(30, 90, 200, 150);

//Пример типа линии и заполнения
Canvas->Pen->Width= 2;
Canvas->Pen->Color= clGreen;//Зелёный цвет пера
Canvas->Pen->Style= psDash;//Выбрана штриховая линия
Canvas->Brush->Color= clGreen;//Зелёный цвет заливки
Canvas->Brush->Style= bsDiagCross;//Ромбическая клетка
Canvas->Rectangle(250, 30, 420, 150);
```

15.6.2. Прямоугольник

Метод *Rectangle* применялся в предшествующем пункте. В этом методе вместо четырёх параметров (координат диагональных углов прямоугольника) можно использовать только один – структуру типа *TRect*, поля которой определяют положение главной диагонали задаваемой прямоугольной области. В следующем фрагменте кода верхняя часть флага из предшествующего приложения *Флаг* реализуется с использованием упомянутого параметра типа *TRect*.

```
//Верхняя часть флага
Canvas->Brush->Color= clBlue;
TRect Rct;//Структура

//Координаты прямоугольной области
Rct.Top= 30;
Rct.Left= 30;
Rct.Bottom= 90;
Rct.Right= 200;
Canvas->Rectangle(Rct);
```

Построить прямоугольник можно при помощи ещё двух методов. Закрашенный в цвет кисти прямоугольник порождается методом *FillRect*, а метод *FrameRect* цветом кисти вычерчивает (толщиной в один пиксель) *только* контур прямоугольника. У этих методов предусмотрен *лишь один* параметр типа *TRect*. Значения полей этой структуры можно задать как прежним способом, так и при

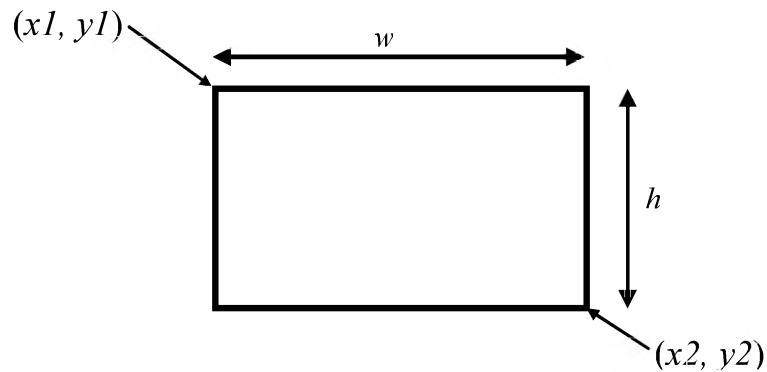


Рис. 15.7. Иллюстрация назначений параметров методов *Rect* и *Bounds*

помощи функций *Rect*(x_1, y_1, x_2, y_2) или *Bounds*(x_1, y_1, w, h) (*bounds* – границы). В функции *Rect* требуется указывать значения координат рубежных (граничных) точек главной диагонали прямоугольника (x_1, y_1, x_2, y_2), а в функции *Bounds* вместо координат нижней точки этой диагонали x_2 и y_2 приводится ширина и высота h прямоугольника (измеряются в пикселях). Более наглядно значения упомянутых параметров поясняет рис. 15.7.

В следующем фрагменте программы верхняя часть прежнего флага выполнена с применением функций *Rect*() и метода *FillRect*, посередине этой части помощью метода *TextRect*() написано название страны.

У метода *TextRect*() имеется 4 параметра. Первый из них определяет местоположение и габариты прямоугольной области, внутри которой он выводит текст, являющийся его четвёртым параметром. Второй и третий аргументы метода – это координаты (по горизонтали и вертикали) верхнего левого угла заднего плана первой литеры (на рис. 15.8 – это буква У) выводимого текста относительно левого верхнего угла упомянутой выше прямоугольной области (обозначенный первым параметром).

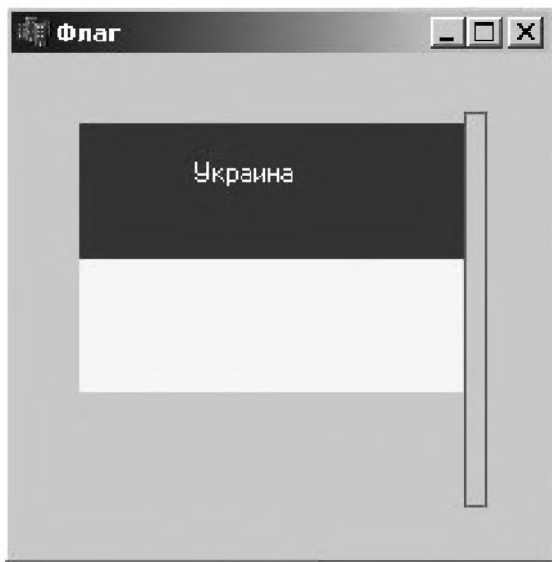
При помощи функции *Bounds* и метода *Rectangle* строится нижняя часть лотнища флага. Его древко вычерчено методом *FrameRect*. Интерфейс приложения, рисующего флаг с древком и надписью, показан на рис. 15.8.

```
//Флаг Украины
Canvas->Pen->Style= psClear;

//Верхняя часть флага
Canvas->Brush->Color= clBlue;
TRect Rct;//Структура

//Координаты прямоугольной области
Rct= Rect(30, 30, 200, 90); //Задание полей структуры Rct
Canvas->FillRect(Rct);
Canvas->Font->Color= clRed; //Цвет надписи
Canvas->TextRect(Rct, 80, 45, "Украина");

//Нижняя часть флага
Canvas->Brush->Color= clYellow;
Rct= Bounds(30, 90, 170, 60); //Иной способ инициализации
Canvas->Rectangle(Rct);
```

Рис. 15.8. Новый интерфейс приложения *Флаг*

```
//Древко флага
Canvas->Brush->Color= clGreen;
Rct= Rect(200, 25, 210, 200);
Canvas->FrameRect(Rct);
```

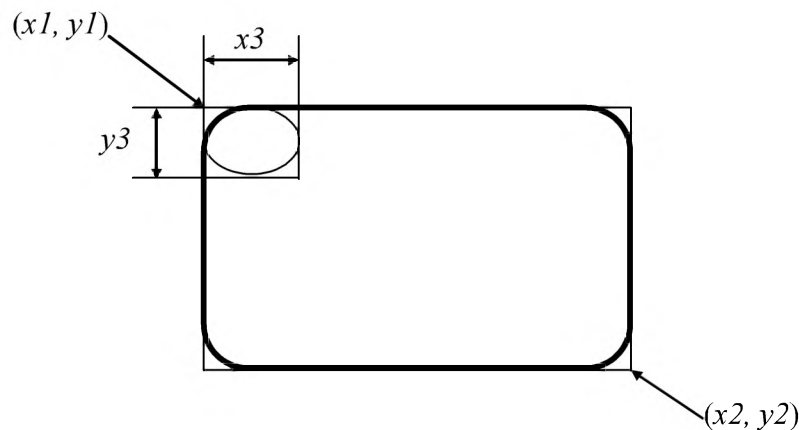
Прямоугольник со скруглёнными углами вычерчивается методом *RoundRect(x1, y1, x2, y2, x3, y3)*. Назначение параметров этого метода поясняет рис. 15.9.

Если перед вызовом метода вычерчивания какой-либо геометрической фигуры *не указывать* значения параметров карандаша и кисти, то (по умолчанию) вычерчивание осуществляется толщиной в один пиксель чёрным цветом, а заливка выполняется белым цветом.

15.6.3. Эллипс

Вывод эллипса или окружности осуществляется методом *Ellipse(x1, y1, x2, y2)*. Назначения его параметров поясняет рис. 15.10. Вы видите, что параметр $x1, y1, x2, y2$ определяют координаты рубежных точек главной диагонали прямоугольника, описывающего эллипс (или квадрата, описывающего окружность).

В методе *Ellipse* вместо четырёх числовых параметров можно использовать один параметр типа *TRect*. Так же, как и в методах *Rectangle*, *FillRect* и *FrameRect*

Рис. 15.9. Иллюстрация назначений параметров метода *RoundRect*

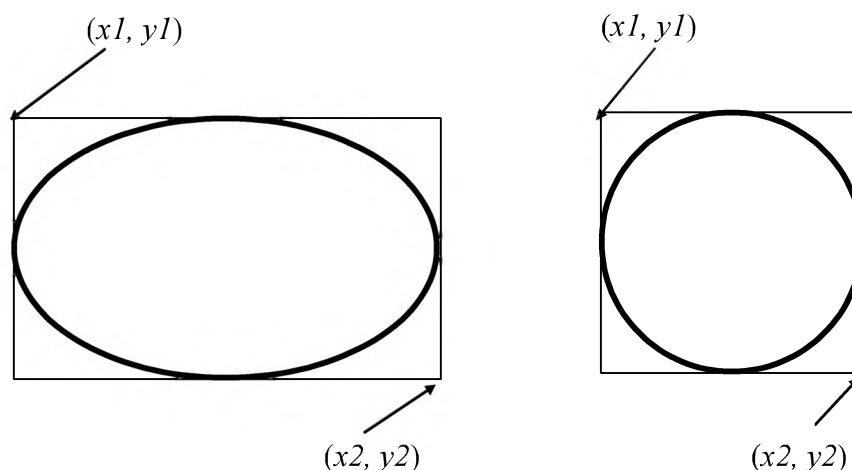


Рис. 15.10. Иллюстрация назначений параметров метода *Ellipse*

упомянутый параметр разрешается задавать тремя способами: непосредственно, определять поля структуры, применять функцию *Rect* либо функцию *Bounds*.

15.6.4. Дуга

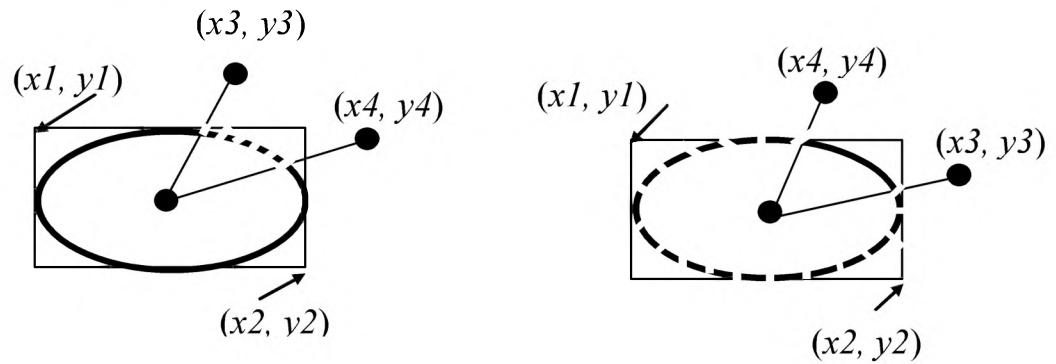
Дуга – это часть эллипса (окружности), она выводится методом *Arc*($x1, y1, x2, y2, x3, y3, x4, y4$). Первые 4 параметра метода ($x1, y1, x2, y2$) определяют эллипс, частью которого является выводимая дуга. Начальная (или конечная) точка дуги – это точка пересечения контура эллипса и отрезка прямой, проведенной между центром эллипса и точкой с координатами ($x3, y3$) (или ($x4, y4$)). Метод *Arc* вычерчивает дугу против часовой стрелки – от начальной до конечной точки. Значения параметров метода поясняет рис. 15.11.

15.6.5. Сектор

Вычерчивание сектора эллипса (или круга) осуществляет метод *Pie*($x1, y1, x2, y2, x3, y3, x4, y4$) (пирог). Параметры этого метода точно такие же, как и в методе *Arc*. Сектор также как и дуга вычерчивается против часовой стрелки, начиная от точки, заданной точкой пересечения контура эллипса и отрезка прямой, проведенной из центра эллипса до точки с координатами ($x3, y3$). Различие в работе методов *Pie* и *Arc* заключается лишь в том, что метод *Pie* вычерчивает замкнутую фигуру, внутренность которой можно залить произвольным цветом (см. рис. 15.12).

Как видно из рисунка, в методе *Pie* вычерчиваются ещё и *радиусы* граничных точек дуги, образующей сектор. Код функции *FormPaint*, которая выполняет построения, приведенные на рис. 15.12, показан ниже. Для получения правой части рисунка следует закомментировать предпоследнюю строку и убрать символы «//» у последней строки приводимых операторов.

```
Canvas->Pen->Width= 5;
Canvas->Pen->Color= clGreen;
Canvas->Brush->Color= clYellow;
Canvas->Pie(30, 30, 200, 150, 130, 30, 200, 60);
//Canvas->Pie(30, 30, 200, 150, 200, 60, 130, 30);
```

Рис. 15.11. Иллюстрация назначений параметров метода *Arc*

15.6.6. Линия

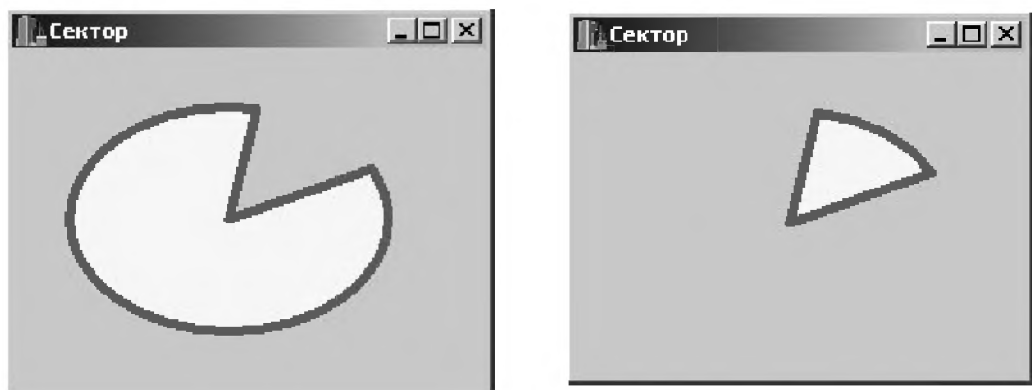
Вычерчивание отрезка прямой из текущей точки расположения карандаша заданную точку (x, y) осуществляет метод *LineTo*(x, y). Начальная точка местоположения карандаша (x_0, y_0) задаётся методом *MoveTo*(x_0, y_0). Вывести вертикальный отрезок с использованием упомянутых методов можно, например, таким операторами:

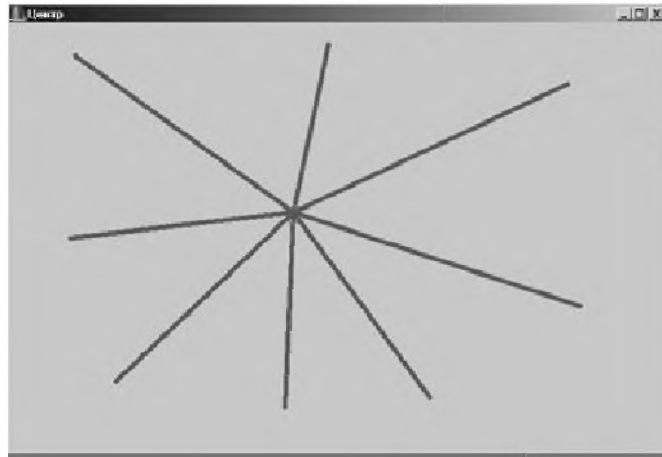
```
Canvas->MoveTo(30, 30);
Canvas->LineTo(30, 200);
```

Следующая учебная программа *Центр* проводит отрезок прямой от позиции формы, где произведен щелчок мышью, до точки формы с координатами (30, 200). Функция *FormMouseDown* обрабатывает событие *OnMouseDown*, связанное с отпусканием кнопки мыши при щелчке по форме.

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
                                     TShiftState Shift, int X, int Y)
{
    Canvas->Pen->Width= 5;
    Canvas->Pen->Color= clGreen;
    Canvas->MoveTo(X, Y); //Текущие координаты щелчка
    Canvas->LineTo(300, 200);
} //FormMouseDown
```

Интерфейс этого приложения показан на рис. 15.13.

Рис. 15.12. Интерфейсы приложения *Сектор* при различных параметрах метода *Pie*

Рис. 15.13. Интерфейсы приложения *Центр*

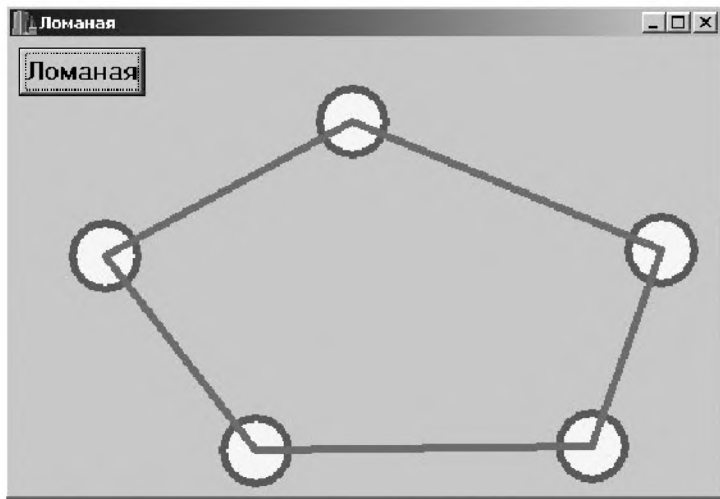
15.6.7. Ломаная линия

При помощи карандаша построение ломаной линии выполняет метод *Polyline(p, in)*. Первый его параметр – это массив структур типа *TPoint*, он содержит координаты узловых точек ломаной линии (точек излома), вторым параметром является количество звеньев ломаной линии (на единицу меньше числа узловых точек). Этот метод последовательно соединяет точки, координаты которых находятся в массиве *p*: нулевую с первой, первую со второй и т. д. Если в последней строке этого массива указать координаты начальной точки, то будет вычерчена *замкнутая* ломаная линия.

Для знакомства с этим методом изучите приложение *Ломаная*. Код приложения приведен ниже. Итоговая картинка интерфейса этой программы показана на рис. 15.14. Фабула работы программы такова. Вначале на пустой форме пользователь щелчками мыши порождает кружки с зелёным контуром и жёлтым заполнением, они возникают в местах щелчков при помощи функции *FormMouseDown*. После появления пятого кружка всплывает диалоговое окно, в котором предлагается нажать по кнопке с заголовком *Ломаная*. В результате её нажатия центры порождённых кружков последовательно соединяются отрезками прямых красного цвета, образуется замкнутая ломаная линия из пяти звеньев.

```
//Глобальные константы и переменные
const int in= 5, //Максимальное число кружков
        D= 45; //Диаметр кружков в пикселях
int m= 0; //Текущий номер кружка
//Координаты начала, точек излома и конца ломаной
TPoint p[in]; //Структура

void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton
Button, TShiftState Shift, int X, int Y)
{
    Canvas->Pen->Width= 5;
    Canvas->Pen->Color= clGreen;
    Canvas->Brush->Color= clYellow;
    if (m< in)
    {
        Canvas->Ellipse(X, Y, X + D, Y + D);
```

Рис. 15.14. Интерфейсы приложения *Ломаная*

```

    p[m].x= X + D/2;
    p[m].y= Y + D/2;
    m++;
} //if
else
{
    ShowMessage("Теперь нажмите кнопку \"Ломаная\"");
    p[m].x= p[0].x;
    p[m].y= p[0].y;
} //else
} //FormMouseDown

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Canvas->Pen->Color= clRed;
    Canvas->Polyline(p, in);
} //Button1Click

```

15.6.8. Многоугольник

Многоугольник отличается от *замкнутой* ломаной линии лишь тем, что в нём осуществляется *заливка* его внутренней части. Рисует многоугольник метод *Polygon(p, k)*, где *p* – массив структур типа *TPoint*, содержащий координаты всех вершин многоугольника, *k* – номер *последней* вершины (на единицу меньше их общего числа, поскольку отсчёт начинается с нуля). Многоугольник создаётся отрезками прямых линий, *последовательно* соединяющих его вершины. Вначале отрезок прямой линии проводится между вершинами, координаты которых находятся в нулевой и первой ячейках массива структур *p*, затем – между вершинами, данные о которой содержатся в первой и второй ячейках упомянутого массива и т. д. Последняя и начальная вершины также связываются отрезком прямой линии. Фабула работы и интерфейс учебной программы аналогичны приложению *Ломаная*. Отличие интерфейсов состоит лишь в том, что внутренняя часть многоугольника *заливается* жёлтым цветом, да на заголовках формы и командной кнопки – надпись *Многоугольник*. Текст приложения приведен ниже.


```
//Глобальные константы и переменные
const int in= 5, //Максимальное число кружков
        D= 45; //Диаметр кружков
int m= 0; //Текущий номер кружка
TPoint p[in]; //Координаты вершин многоугольника

void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
TShiftState Shift, int X, int Y)
{
    Canvas->Pen->Width= 5;
    Canvas->Pen->Color= clGreen;
    Canvas->Brush->Color= clYellow;
    if (m< in)
    {
        Canvas->Ellipse(X,Y,X + D,Y + D);
        p[m].x= X + D/2;
        p[m].y= Y + D/2;
        m++;
    } //if
    else
        ShowMessage("Теперь нажмите кнопку \"Многоугольник\"");
} //FormMouseDown

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Canvas->Pen->Color= clRed;
    Canvas->Polygon(p, in - 1);
} //Button1Click
```

15.6.9. Текст

Без вывода текста в графических приложениях обойтись практически нельзя. Текст используется, например, при обозначении координатных осей и значений её шкалы, для наименования кривых графика и во многих других случаях. В п. 15.6.2 использовалась функция *TextRect()*, которая позволяет выводить текст в заданном месте прямоугольной области. Показать строку текста на поверхности графического объекта можно также при помощи метода *TextOutA(x, y, sT)*. Его два параметра – это координаты точки графической поверхности, с которой начинается текст (координаты левого верхнего угла прямоугольного окошка заднего плана первой литеры текста), параметр *sT* (типа *String*) – определяет тот текст, который должен появиться на экране.

Многие параметры текста можно, конечно, установить при помощи *Инспектора Объектов*, однако программа становится значительно более документированной и понятной (без дополнительных комментариев), если эти параметры определяются в самом коде программы. Среди очень полезных методов класса *TFont* имеются методы *TextHeight(sT)* и *TextWidth(sT)*. Они определяют соответственно высоту и ширину заданного текста с ранее установленными (любым из способов) параметрами. С их использованием текст можно расположить в заданном месте поверхности графического объекта. Для прозрачности заднего плана текста перед его выводом устанавливается стиль *bsClear* кисти:



Рис. 15.15. Интерфейсы приложения *Текст*

```
Canvas->Brush->Style= bsClear;
```

В противном случае надпись появляется на фоне полосы с цветом заднего плана шрифта (если цвет фона поверхности, на которой выводится текст, не совпадает с цветом заднего плана текста).

Интерфейс иллюстративного приложения *Текст* показан на рис. 15.15. Он представляет собой две полосы разных цветов, габариты которых одинаковы, посередине интерфейса чёрным цветом шрифтом размером 84 пункта выведено слово *Графика*. Более приятным цветом для этого интерфейса является зелёный (*clGreen*), однако полиграфические ограничения книги диктуют нам чёрный цвет.

При одновременном выборе *разных* размеров шрифта в *Инспекторе Объектов* и в коде приложения, приоритет за установками, осуществлёнными в тексте программы. Это касается *всех* свойств и подсвойств *Инспектора Объектов*.

Установка выбранных параметров стиля текста (при помощи соответствующих констант) выполняются с использованием метода *TFontStyles()*, обеспечивающим помещение их в одноимённое множество параметров текста.

```
Canvas->Font->Style= TFontStyles()<<fsBold<<fsItalic;
```

Теперь изучите текст приложения, разъяснения к нему приведены выше.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    String sT= "Графика";//Выводимый текст
    TRect Fon_Rect;
    int x, y;//Координаты начала текста

    //Верхняя половина окна имеет синий цвет
    Fon_Rect= Rect(0, 0, ClientWidth, ClientHeight/2);
    Canvas->Brush->Color= clBlue;
    Canvas->FillRect(Fon_Rect);

    //Нижняя половина окна — жёлтого цвета
    Fon_Rect= Rect(0, ClientHeight/2, ClientWidth, ClientHeight);
    Canvas->Brush->Color= clYellow;
```

```

Canvas->FillRect(Fon_Rect);
Canvas->Font->Name= "Time New Roman";//Название шрифта
Canvas->Font->Size= 84;//Размер шрифта
Canvas->Font->Style= TFontStyles()<<fsBold<<fsItalic;

//Размещение текста по центру интерфейса
x= (ClientWidth-Canvas->TextWidth(sT))/2;
y= (ClientHeight-Canvas->TextHeight(sT))/2;

//Запрет закрашивания области вывода текста
Canvas->Brush->Style= bsClear;
Canvas->Font->Color= clBlack;//Более приятный цвет clGreen
Canvas->TextOutA(x, y, sT);// Вывод текста
} //FormPaint

```

При программировании может сложиться ситуация, когда требуется вывести какое-то сообщение за текстом, длина и местоположение которого во время разработки приложения ещё *неизвестны* (определяются событиями в ходе работы программы). В таком случае применяются методы *PenPos.x* и *PenPos.y*, определяющие координаты конца исходного (текущего) текста. Здесь *x* и *y* – фактически являются координатами для вывода первой литеры за уже существующим (имеющимся) текстом.

Модернизируем предшествующую программу так, чтобы на её интерфейсе в конце текста стоял восклицательный знак. Воспользуемся следующим фрагментом кода (его необходимо вписать после последнего оператора функции *FormPaint*):

```

//Вывод нового текста после уже существующего текста
//неизвестной длины
Canvas->TextOutA(Canvas->PenPos.x, Canvas->PenPos.y, "!");

```

15.7. Приложение Пособие

Для накопления навыков разработки графических программ познакомимся с приложением *Пособие*, оно позволяет исследовать поведение графиков параболы и окружности при различных значениях параметров их функциональных зависимостей.

Как известно, парабола определяется уравнением $y = Ax^2 + Bx + C$. При $A = 0$ оно упрощается к уравнению прямой.

Уравнение окружности с радиусом R имеет вид: $(x - a)^2 + (y - b)^2 = R^2$. При $a = 0$ и $b = 0$ центр окружности совпадает с началом координат. В этом случае проекция любой точки окружности на оси координат зависит от радиуса окружности R и угла α (см. рис 15.16 а). Если же координаты центра окружности смещены относительно начала координат ($a \neq 0, b \neq 0$), то проекции радиус-вектора точки окружности выражаются зависимостями: $x = a + R \cos \alpha, y = b + R \sin \alpha$ (см. рис.15.16 б).

Упомянутые сведения способствуют пониманию приведенного ниже кода программы. В этом коде константы параболы называются также как и в её уравнении, а параметры центра окружности имеют несколько другие имена: *AA, BB*. Для обозначения радиуса выбрана та же буква R , что и в уравнении окружности.

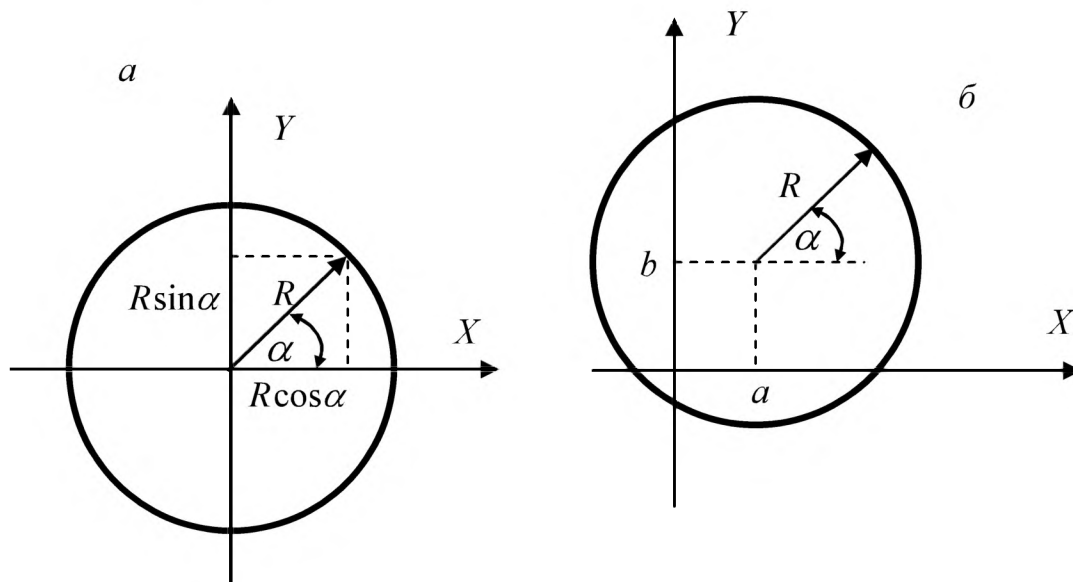


Рис. 15.16. Проекции радиус-вектора произвольной точки окружности

Интерфейсы приложения для двух различных наборов коэффициентов упомянутых уравнений показаны на рис.15.17.

При запуске приложения выводятся зависимости (верхняя панель рис. 15.17), коэффициенты для которых присвоены соответствующим глобальным переменным в качестве стартовых значений (они введены в коде программы). На нижней панели рисунка использованы такие коэффициенты: $A=0$, $B=0,5$, $C=5$, $AA=12$, $BB=5$, $R=15$ (их следует ввести с использованием горизонтального меню приложения *Пособие*). При нажатии кнопки *Показать* появляются зависимости, коэффициенты для которых установлены при помощи соответствующих пунктов (и их подпунктов) горизонтального меню. Как разрабатывается горизонтальное меню описано в п. 9.5.1.

Пункты подменю для ввода коэффициентов имеют соответствующие имена: $Vvod_A$, $Vvod_B$, $Vvod_C$, $Vvod_AA$, $Vvod_BB$, $Vvod_R$. При выборе этих подпунктов появляются окна, служащие для ввода коэффициентов уравнений. Имена окон ясно подсказывают, для ввода каких коэффициентов они предназначены: Vv_A , Vv_B , Vv_C , Vv_AA , Vv_BB , Vv_R . С целью уменьшения объёма листинга приводится только функция $Vvod_AClick$, обеспечивающая ввод коэффициента A . Функции для ввода других коэффициентов аналогичны показанной и поэтому не приводятся.

Выводу графиков предшествует начертание при помощи функции Osi осей координат, начало которых размещается в центре интерфейса с координатами X_Centr и Y_Centr . В этой функции использован переход от графических координат (с началом в верхнем левом углу формы) к математическим (с началом в центре формы). При этом 10 пикселям графических координат соответствует единица математических координат. Название осей и их разметка осуществляются при помощи метода $TextOutA$.

Нажатием кнопки *Стереть* осуществляется устранение с экрана выведенных ранее кривых. С этой целью в функции $SteretjClick$ прежние кривые выводятся цветом заднего плана ($clSilver$ – серебристый), применяемым для изначальной

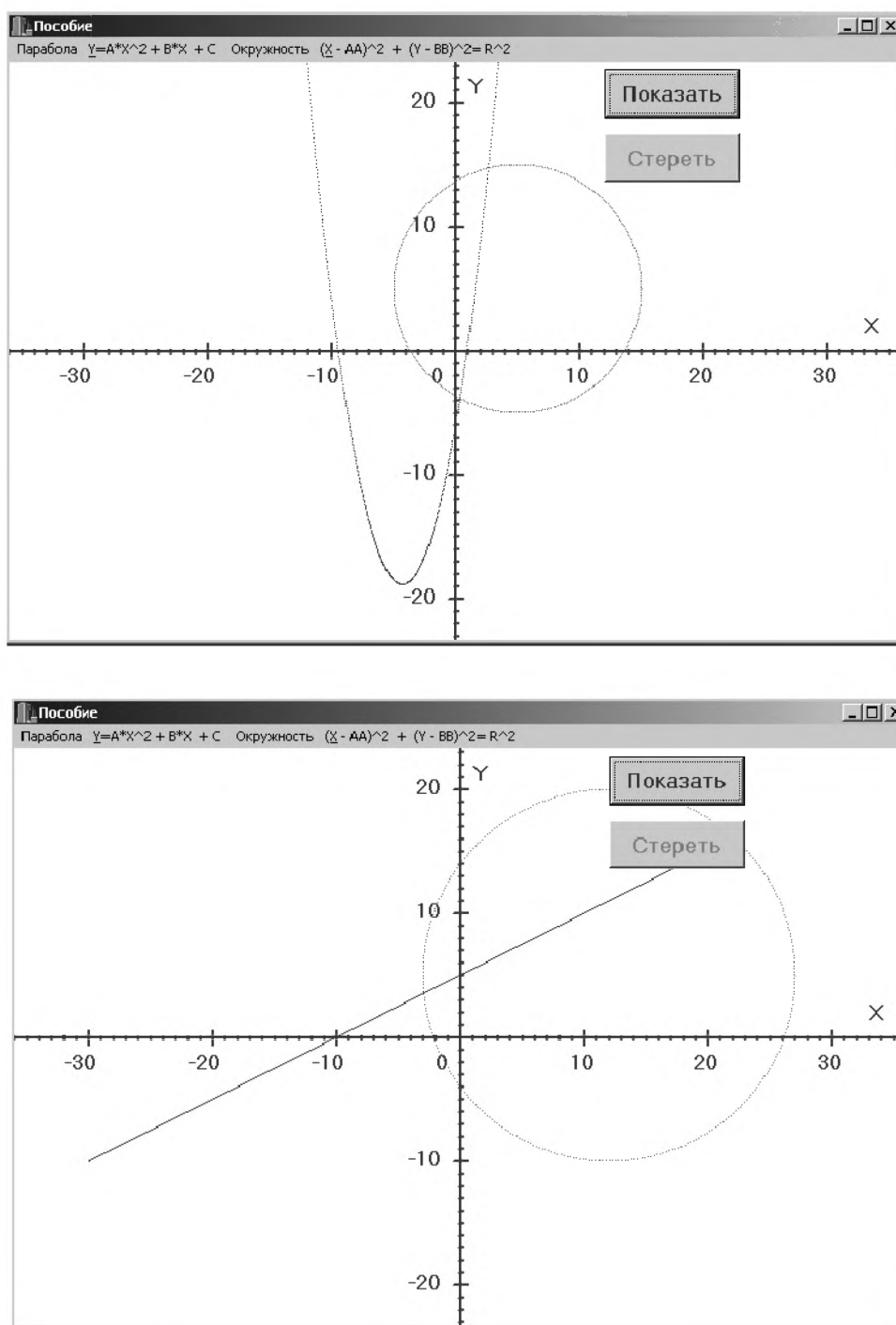


Рис. 15.17. Интерфейсы приложения *Пособие*.

Верхняя панель – использованы параметры зависимостей, установленные по умолчанию;
нижняя панель – значение параметров зависимостей приведены в тексте

закраски поверхности формы.

В приложении используется тип *TColor* для задания переменных перечислимого типа, которым в ходе выполнения программы назначаются различные константы цвета. Эти цветовые константы являются константами перечисления (см. раздел 13.1) с именем *TColor*. Основные возможности типа *TColor* излагаются в ходе настоящего урока.

```
#include <math.h>
int X_Centr, Y_Centr;
float A= 0.7, B= 6, C=- 6, AA= 5, BB= 5, R= 10, x,
```

```
y, dx= 0.03;
//Цвет параболы и окружности
    TColor Cvet_Par= clBlue, Cvet_Okr= clRed;

void Osi(void)
{
    int ii, nn;
    Form1->Canvas->Pen->Width= 2;
    Form1->Canvas->Pen->Color= clPurple;//Сиреневый цвет

    //Вертикальная ось
    Form1->Canvas->MoveTo(X_Centr, Form1->ClientHeight);
    Form1->Canvas->LineTo(X_Centr, 0);

    //Название вертикальной оси
    Form1->Canvas->Brush->Style= bsClear;
    Form1->Canvas->Font->Color= clPurple;
    Form1->Canvas->TextOutA(X_Centr + 10, 10, "Y");

    //Горизонтальная ось
    Form1->Canvas->MoveTo(Form1->ClientWidth, Y_Centr);
    Form1->Canvas->LineTo(0, Y_Centr);

    // Название горизонтальной оси
    Form1->Canvas->TextOutA(X_Centr*2-30, Y_Centr-30, "X");

    //Засечки горизонтальной оси
    nn= X_Centr/10;//Масштаб: 10 пикселей равны единице
    for (int ii= -nn, d; ii<= nn; ii++)
    {
        if (fmod(ii, 10))
            d= 2;
        else
        {
            d= 6;
            if (ii> 0)
                Form1->Canvas->TextOutA(X_Centr+10*ii-12, Y_Centr+10, ii);
            else
                Form1->Canvas->TextOutA(X_Centr+10*ii-20, Y_Centr+10, ii);
        }//else_1
        Form1->Canvas->MoveTo(X_Centr + 10*ii, Y_Centr - d);
        Form1->Canvas->LineTo(X_Centr + 10*ii, Y_Centr + d);
    }//for

    //Засечки вертикальной оси
    nn= Y_Centr/10;//Масштаб: 10 пикселей равны единице
    for (int ii= -nn, d; ii<= nn; ii++)
    {
        if (fmod(ii, 10))
            d= 2;
        else
        {
            d= 6;
            if (ii> 0)
                Form1->Canvas->TextOutA(X_Centr-d*7, Y_Centr+10*ii-12, -ii);
            if (ii< 0)//else
```

```

        Form1->Canvas->TextOutA(X_Centr-d*6, Y_Centr+10*ii-12, -ii);
    } //else_1
    Form1->Canvas->MoveTo(X_Centr -d, Y_Centr - 10*ii);
    Form1->Canvas->LineTo(X_Centr +d, Y_Centr - 10*ii);
} //for
} //Osi

void Parabola(float Aa, float Bb, float Cc)
{
    x= -30;
    do
    {
        y= Aa*pow(x, 2) + Bb*x + Cc;
        Form1->Canvas->Pixels[X_Centr + 10*x][Y_Centr - 10*y]= Cvet_Par;
        x += dx;
    } //do
    while (x<= 30);
} //Parabola

void Okrygnostj(float AAa, float BBb, float Rr)
{
    int ii;
    float Alfa;
    for (int ii= 0; ii<= 360; ii++)
    {
        Alfa= ii*M_PI/180; //Переход к радианной мере
        Form1->Canvas->Pixels[X_Centr+10*(AAa+Rr*cos(Alfa))][
            Y_Centr-10*(BBb+Rr*sin(Alfa))]= Cvet_Okr;
    } //for
} //Okrygnostj

void __fastcall TForm1::Vvod_AClick(TObject *Sender)
{
    Vv_A->Visible= true;
} //Vvod_AClick

```

Функции *Vvod_BClick*, *Vvod_CClick*, *Vvod_AAClick*, *Vvod_BBClick* и *Vvod_RClick* алогичны функции *Vvod_AClick*.

```

void __fastcall TForm1::FormPaint(TObject *Sender)
{
    X_Centr= Form1->ClientWidth/2;
    Y_Centr= Form1->ClientHeight/2;
    Osi();
    Parabola(A, B, C);
    Okrygnostj(AA, BB, R);

    Vv_A->Visible= false; //Устраняем с экрана окна ввода
    Vv_B->Visible= false;
    Vv_C->Visible= false;

    Vv_AA->Visible= false;
    Vv_BB->Visible= false;
    Vv_R->Visible= false;
} //FormPaint

void __fastcall TForm1::PokazClick(TObject *Sender)
{

```

```
Cvet_Par= clBlue;
Cvet_Okr= clRed;

A= StrToFloat(Vv_A->Text);
B= StrToFloat(Vv_B->Text);
C = StrToFloat(Vv_C->Text);

AA= StrToFloat(Vv_AA->Text);
BB= StrToFloat(Vv_BB->Text);
R= StrToFloat(Vv_R->Text);

TForm1::FormPaint(Form1);
} // PokazClick

void __fastcall TForm1::SteretjClick(TObject *Sender)
{
    Cvet_Okr= clSilver;
    Cvet_Par= clSilver;
    TForm1::FormPaint(Form1);
} // SteretjClick
```

15.8. Мультипликация с использованием графических примитивов

Мультипликацию с использованием графических примитивов изучим на примере приложения *Полёт стрел*, в котором подвижное изображение создаётся первыми *тремя* способами, описанными в начале настоящего урока. Кратко напомним, как осуществляется движение. Во *всех* способах мультипликации фрагмент изображения выводится каким-либо цветом или даже разными цветами, а затем, через небольшой интервал времени, этот фрагмент стирается. Далее вывод и стирание фрагмента производятся в несколько смещённом положении. Траектория движения может быть произвольной.

В первом способе стирание фрагмента осуществляется так: цвет контуров и заливок (если они есть) всех его частей устанавливается совпадающим с цветом заднего плана. Во втором способе стирание фрагмента изображения, выбранного для движения, выполняется по-иному: на весь такой фрагмент накладывается прямоугольник (сектор, эллипс и др.) цвет контура и заливки которого совпадает с цветом заднего плана. Габариты стирающего прямоугольника (примитива) должны быть на несколько пикселей больше габаритов стираемого фрагмента. В третьем способе для стирания применяется режим пера *mpNotXor*.

Фабула программы-видеоролика такова. Стрела красного цвета (появляющаяся слева экрана) летит горизонтально в сторону мишени, расположенной в правой части окна. После попадания стрелы в мишень вылетает другая, зелёная, стрела (несколько ниже прежней), она также вонзается в мишень. Далее всё повторяется для стрел других цветов. В конце мультфильма в мишени будут торчать 9 стрел *разного* цвета (см. рис. 15.18). Приложение *Полёт стрел* иллюстрирует применение 9 стандартных цветов перечисления *TColor*. В справке (*Help* в *Builder*) или в [3] указывается типовое назначение этих и других цветовых кон-

стант. Набор допустимых цветов формы (и их цветовые константы) указан в *Инспекторе Объектов* в свойстве *Color*. Любой из этих цветов применим также и для других целей.

Приложение содержит несколько пользовательских функций. Функция *Mishen* выводит мишень: концентрические эллипсы сиреневого цвета, расположенные в правой части экрана, их вертикальная ось превышает горизонтальную в 3 раза.

Функция *Strela(int ix, int idy)* рисует простейшее изображение стрелы. Параметры *ix* и *idy* изменяют местоположение стрелы соответственно по горизонтали и вертикали.

Функция *Pysk_Strelu(int idy, TColor Cvet_Strelu)* запускает стрелу заданного цвета (*Cvet_Strelu*) с выбранной высотой полёта (устанавливается аргументом *idy* относительно центра экрана по вертикали *Y_Centr*). Изменением формального параметра *ix* в функции *Strela* обеспечивается горизонтальное перемещение стрелы. Скорость такого перемещения регулируется функцией *Sleep* при помощи её единственного аргумента, измеряемого в миллисекундах. Эта функция задерживает полёт стрелы на заданный интервал времени.

Применение функции *Zapysk_Strl* осуществляет последовательный запуск *всех* стрел на ряде высот и с разной окраской. Ниже приводится код приложения, в котором реализуется первый способ мультипликации.

```
int X_Centr, Y_Centr; //Глобальные переменные
TColor Cvet; //файла реализации

void Mishen(void)
{
    const int n= 5, //Число колец мишени
    rx= Form1->ClientWidth/60; //Радиус наименьшего кольца
    X_Centr= Form1->ClientWidth/2;
    Y_Centr= Form1->ClientHeight/2;
    int Centr_Mish= Form1->ClientWidth*0.9; //Центр по оси x
    Form1->Canvas->Pen->Width= 2;
```

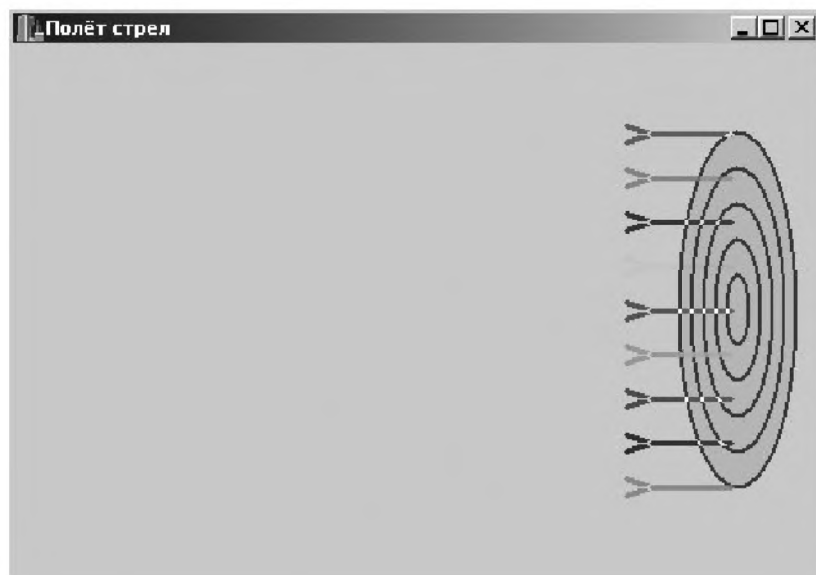


Рис. 15.18. Интерфейсы приложения *Полёт стрел*

```

Form1->Canvas->Pen->Color= clPurple;//clRed;
Form1->Canvas->Brush->Color= clSilver;
for (int ii= n; ii>= 1; ii--)//Вывод эллипсов
    Form1->Canvas->Ellipse(Centr_Mish-rx*ii, Y_Centr-
                          rx*ii*3, Centr_Mish+rx*ii, Y_Centr+rx*ii*3);
} //Mishen

void Strela(int ix,int idy)//idy-смещение стрелы от Y_Centr
{
    const int dl_Strelu= Form1->ClientWidth*0.1;//Длина
    Form1->Canvas->Pen->Width= 3;
    Form1->Canvas->MoveTo(ix, Y_Centr+idy);//Тело стрелы
    Form1->Canvas->LineTo(ix+dl_Strelu, Y_Centr+idy);
    Form1->Canvas->MoveTo(ix-15, Y_Centr+idy-5);//Верхнее
    Form1->Canvas->LineTo(ix, Y_Centr+idy);//перо
    Form1->Canvas->MoveTo(ix-15, Y_Centr+idy+5);//Нижнее
    Form1->Canvas->LineTo(ix, Y_Centr+idy);//перо
} //Strela

void Pysk_Strelu(int idy, TColor Cvet_Strelu)
    // idy-Смещение стрелы от Y_Centr
{
    const int dl_Strelu= Form1->ClientWidth*0.1;//Длина
    for (int ii= 0, ix; ii< 8; ii++)//Движение стрелы
    {
        ix= dl_Strelu + ii*dl_Strelu;
        Form1->Canvas->Pen->Color= Cvet_Strelu;
        Strela(ix, idy);//Рисуем стрелу
        //Задержка выполнения программы, измеряется в мс
        Sleep(200);//0,2 с
        Form1->Canvas->Pen->Color= clSilver;
        Strela(ix, idy);//Стираем стрелу
        if (ii== 7)//Стрела вонзается в мишень
        {
            Form1->Canvas->Pen->Color= Cvet_Strelu;
            Strela(ix, idy);
        } //if
    } //for
} //Pysk_Strelu

void Zapysk_Strl(void)
{
    const int ihy= Y_Centr/6;// Расстояние между стрелами
    for (int ii= -4, idy; ii<= 4; ii++)
    {
        switch (ii)//Задание различных цветов стрелы
        {
            case -4: Cvet= clRed;//Самая верхняя стрела
                    break;
            case -3: Cvet= clGreen;
                    break;
            case -2: Cvet= clBlue;
                    break;
            case -1: Cvet= clYellow;
                    break;
            case -0: Cvet= clPurple;

```

```
        break;
    case 1: Cvet= clOlive;
        break;
    case 2: Cvet= clMaroon;
        break;
    case 3: Cvet= clNavy;
        break;
    case 4: Cvet= clTeal;
} //switch
idy= ihy*ii; //Задание различных высот полёта стрелы
Pysk_Strelu(idy, Cvet); //Запуск отдельной стрелы
} //for
} //Zapysk_Strl

void __fastcall TForm1::FormPaint(TObject *Sender)
{
    Mishen();
    Zapysk_Strl();
} //FormPaint
```

Приложение, использующее второй способ разработки мультфильмов, отличается лишь функцией *Pysk_Strelu*, её код приведен ниже.

```
void Pysk_Strelu(int idy, TColor Cvet_Strelu)
{
    const int dl_Strelu= Form1->ClientWidth*0.1;
    for (int ii= 0, ix; ii< 8; ii++)
    {
        ix= dl_Strelu + ii*dl_Strelu;
        Form1->Canvas->Pen->Color= Cvet_Strelu;
        if (ii< 7)
        {
            Strela(ix, idy); //Рисуем стрелу
            Sleep(100);
            Form1->Canvas->Pen->Color= clSilver;
            TRect Rct= Rect(ix-18, Y_Centr+idy-7,
                           ix+dl_Strelu+2, Y_Centr+idy+7);
            Form1->Canvas->FillRect(Rct); //Стираем стрелу
        } //if
        else
        {
            //Стрела вонзается в мишень
            Form1->Canvas->Pen->Color= Cvet_Strelu;
            Strela(ix, idy);
        } //else
    } //for
} //Pysk_Strelu
```

Третий способ анимации обеспечивается применением режима *pmNotXor*, который (при повторном выводе фрагмента) изображение замещает (стирает) цветом предшествующего фона.

```
void Pysk_Strelu(int idy, TColor Cvet_Strelu)
{
    const int dl_Strelu= Form1->ClientWidth*0.1; //Длина
    for (int ii= 0, ix; ii< 8; ii++)
    {
```

```

ix= dl_Strelu + ii*dl_Strelu;
Form1->Canvas->Pen->Color= Cvet_Strelu;
//Установка режима стирания предшествующего изображения
Form1->Canvas->Pen->Mode= pmNotXor;
Strela(ix, idy); //Рисуем стрелу
Sleep(100);
Strela(ix, idy); //Стираем стрелу
if (ii== 7)
{
    Form1->Canvas->Pen->Color= Cvet_Strelu;
    Strela(ix, idy); //Стрела долетела до мишени
} //if
} //for
} //Pysk_Strelu

```

15.9. Мультипликация с применением битовых образов

Фабулу прежних приложений осуществим теперь при помощи битовых образов. Битовый образ – это небольшая картинка, которая хранится в динамической памяти компьютера. При помощи методов канвы (в *TForm*, *TImage* и др.) её можно последовательно выводить в *различные* участки канвы, а спустя некоторое время – стирать (заменять фоном). Битовые образы хранятся в файлах с расширением *bmp* – *bit map picture* (битовые матрицы). Название битовый образ – происходит от упомянутого типа файлов. Битовые матрицы используются для хранения растровой графики.

15.9.1. Использование файлов типа bmp

Картинки можно изготовить в графических редакторах, например, *Image Editor* (см. п. 9.5.3), *Borland Resource Workshop* (редактор ресурсов) или *Paint*. Редактор *Image Editor* используется в *следующем* пункте. Сейчас же применим один из самых простых и известных редакторов *MS-Paint* (полагаем, что у читателей имеется опыт работы с ним).

В папке проекта изготовьте 10 файлов: 9 стрел разного цвета и фон. Ниже приводится последовательность действий для изготовления фона.

1. В пункте горизонтального меню *Paint* с именем *Рисунок* активизируйте раздел *Атрибуты* и установите размер холста с габаритами 96x19 пикселей.
2. Заполните холст цветом *clSilver*.
 - 2.1. Скопируйте в буфер обмена изображение пустого интерфейса приложения (с цветом по умолчанию) при помощи команды *Alt+Print Screen*.
 - 2.2. Скопируйте содержимое буфера во временный файл *Paint* и вырежьте из него небольшой участок фона в буфер обмена;
 - 2.3. Вставьте этот участок на упомянутый холст, а затем при помощи инструментов *Выбор цветов* и *Кисть* (или *Заливка*) цветом фрагмента заполните *весь* холст.

2.4. Файл сохраните (в каталоге приложения) под именем *fon.bmp*.

Изготовление файлов (с расширением *bmp*) *strela_1*, *strela_2*,..., *strela_9*: на холсте указанного размера нарисуйте стрелу одного и того же начертания, но разными цветами палитры (по вашему выбору).

В самом начале файла реализации нового варианта приложения (см. ниже) объявляется два указателя на объекты типа *TBitmap* с именами *Fon* и *Strelka*. Динамический объект первого указателя предназначен для хранения изображения фона. Указатель *Strelka* – направлен на динамический объект, в котором *последовательно* запоминаются изображения стрел *разных* цветов. Далее указатели *Fon* и *Strelka* иногда для краткости будем называть объектами. Это, конечно, неправильно, но *все* визуальные объекты *C++Builder* на самом деле являются *указателями* на динамические объекты соответствующих типов, поэтому к таким головоломкам читатели, видимо, привыкли.

Для хранения координат упомянутых прямоугольных областей с фоном и стрелками используются структуры типа *TRect* (подробнее см. п. 15.6.2) с соответствующими именами *Fon_Rect* и *Strel_Rect*. Координаты текущего значения верхнего левого угла фрагмента со стрелой обозначим переменными *ix* и *iy*, а его постоянные ширину и высоту константами *w* и *h*.

Функция *FormCreate* подготавливает приложение к работе: при помощи значения *alClient* свойства формы *Align* интерфейс программы разворачивается на *весь* экран; операция *new* в указатель *Fon* записывает адрес динамического объекта, в который из файла *fon.bmp* загружается изображение фона

```
Fon->LoadFromFile("fon.bmp");
```

После этого создаётся ещё один объект в динамической памяти типа *TBitmap*, однако изображения стрелок в него будут *последовательно* загружаться *лишь* в функции *Zapysk_Strl*, в обсуждаемой функции этому объекту *только* предписывается иметь прозрачный задний план.

В самом конце функции *FormCreate* переменной *ix* назначается стартовое значение. Отрицательное значение этой величины выбрано для того, чтобы стрела *постепенно* выдвигалась с левой границы окна интерфейса (не сразу появлялась на экране). Стартовым значением этой величины регулируется также абсцисса попадания стрел в мишень (передвигается торчащая стрела по горизонтали). При объявлении глобальной переменной *ix* не было назначено её стартовое значение лишь для *удобства описания* работы программы. Мишень рисуется функцией *Mishen*, её код не изменился.

Функция *Pysk_Strelu*, как и ранее, обеспечивает запуск стрелы на выбранной высоте *idy* относительно положения *Y_Centr*. В этой функции для наполнения переменных *Fon_Rect* и *Strel_Rect* используется функция *Bounds* (см. п. 15.6.2). Значения полей структуры *Fon_Rect* – неизменны, поскольку они определяют габариты канвы объекта *Fon*. Поля структуры *Strel_Rect* – это текущее положение прямоугольного фрагмента изображения с летящей стрелой.

Метод *Draw* *выводит* графические изображения *любых* типов на канву произвольного графического объекта:

```
Form1->Canvas->Draw(ix, Y_Centr+idy, Strelka);
```

Здесь *ix* и *Y_Centr+idy* – координаты верхнего левого угла прямоугольной области канвы, куда копируется изображение из графического объекта *Strelka*. В описании метода *Draw* тип последнего параметра является общим абстрактным типом (классом) *TGraphics* (см. п. 14.4.3). Таким образом, приведенная строка кода является примером применения полиморфного параметра функции.

Для *стирания* текущего положения стрелы используется метод *CopyRect*:

```
Form1->Canvas->CopyRect(Strel_Rect, Fon->Canvas, Fon_Rect);
```

Первый параметр *Strel_Rect* устанавливает местоположение прямоугольной области на канве объекта *Form1* для копирования в неё (с целью стирания изображения стрелы) изображения, хранимого во втором параметре *Fon->Canvas* (на канве динамического объекта *Fon*). Третий параметр *Fon_Rect* метода *CopyRect* определяет, с какого участка канвы упомянутого динамического объекта осуществляется копирование.

Местоположение верхнего левого угла фона совпадает с верхним левым углом канвы объекта *Fon*, габариты фона *равны* габаритам стрелы. Однако, в принципе, габариты прямоугольной области с фоном могут существенно превышать габариты того объекта, который предполагается стирать. В третьем параметре метода *CopyRect* можно указать *текущее* положение прямоугольного фрагмента фона, который необходимо вырезать для стирания изображения стрелы в *конкретном* положении на форме (в заданный момент на траектории движения стрелы). Это применяется, когда стирание фрагмента осуществляется на *непостоянном* фоне (задний план изменяется в зависимости от местоположения перемещающегося фрагмента).

После попадания стрелы в мишень метод *CopyRect* не применяется, поскольку согласно фабуле мультфильма стрела должна остаться торчащей в мишени (поэтому она не стирается на конечном участке полёта). Чтобы очередная стрела, запускаемая функцией *Pysk_Strelu*, снова вылетела из-за левой границы окна формы, переменной *ix* вновь назначается прежнее стартовое значение (*ix = -30;*).

Функция *Zapysk_Strl* также как и в предшествующих приложениях запускает стрелы на разных высотах экрана. С этой целью для каждой стрелы вначале формируется её смещение относительно центра, а затем в динамический объект загружается её цветное изображение из соответствующего файла, имя которого формируется согласно номеру запускаемой стрелы. Запуск стрелы с конкретными параметрами осуществляет функция *Pysk_Strelu*.

Последней в приведенном коде является функция *FormDestroy* (обработчик события формы *OnDestroy*), следуя хорошему стилю программирования, в ней уничтожаются все ранее порождённые динамические объекты. Операция *delete* автоматически вызывает метод *Free()* уничтожаемого ею динамического объекта. Эта функция вначале выяснит, был ли объект создан (не равен ли указатель на объект *NULL*), не была ли ранее уже освобождена выделенная под объект память и только если объект порождён и ещё не уничтожен, вызывается его деструктор. Такая цепочка операций предотвращает нежелательные исключительные ситуа-

ции при порождении и уничтожении динамических объектов *в ходе* выполнения программы.

Однако если не уничтожить динамические объекты *Fon* и *Strelka* в конце работы учебного приложения, то это сделает форма при своём закрытии.

```
Graphics::TBitmap *Fon,*Strelka;
//Координаты фрагмента изображения со стрелкой и фоном
TRect Strel_Rect, Fon_Rect;//Структуры
int X_Centr, Y_Centr,ix;//ix- абсциса левого края фрагмента
    Centr_Mish,//Местоположение центра мишени по оси x
    rx;//Радиус наименьшего кольца мишени
const int w= 96, h= 19,//96x19 - габариты фрагмента
    n= 5;//Число колец мишени

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Form1->Align= alClient;//Интерфейс развёрнут на весь экран
    //Загрузка фона из - файла
    //Создаём объект для хранения графики
    Fon= new Graphics::TBitmap();
    //Запоминаем в нём изображение фона
    Fon->LoadFromFile("fon.bmp");
    //Создаём ещё один объект для хранения графики
    Strelka= new Graphics::TBitmap();
    //Задний план в будущем рисунке будет прозрачным
    Strelka->Transparent= true;
    //Исходное положение левого конца стрелы
    ix= -30;//Для управления координатами острия
}

void Mishen(void)
{
    rx= Form1->ClientWidth/60;
    X_Centr= Form1->ClientWidth/2;
    Y_Centr= Form1->ClientHeight/2;
    Centr_Mish= Form1->ClientWidth*0.9;//Центр мишени по x
    Form1->Canvas->Pen->Width= 2;
    Form1->Canvas->Pen->Color= clPurple;
    Form1->Canvas->Brush->Color= clSilver;
    for (int ii= n; ii>= 1; ii--)
        Form1->Canvas->Ellipse(Centr_Mish-rx*ii, Y_Centr-rx*ii*3,
            Centr_Mish+rx*ii, Y_Centr+rx*ii*3);
}

void Pysk_Strelu(int idy)
{
    while (ix<= Centr_Mish-rx*n-w)//Полёт до мишени
    {
        Strel_Rect= Bounds(ix, Y_Centr+idy, w, h);
        Fon_Rect= Bounds(0, 0, w, h);
        //Рисуем стрелу
        Form1->Canvas->Draw(ix, Y_Centr+idy, Strelka);//aaa
        Sleep(100);//Задержка выполнения программы
        //Стираем стрелу
        Form1->Canvas->CopyRect(Strel_Rect, Fon->Canvas, Fon_Rect);
        ix += w;//w - шаг перемещения стрелы
    }
}
```

```

    if (ix> Centr_Mish-rx*n-w)//Стрела торчит в мишени
        Form1->Canvas->Draw(ix, Y_Centr+idy, Strelka);
    }//while
    ix= -30;//Возврат аргумента в исходное состояние
} //Pysk_Strelu

void Zapysk_Str1(void)
{
    const int ihy= Y_Centr/6;// Высотный шаг между стрелами
    for (int ii= -4, idyy, nom= 1; ii<= 4; ii++)
    {
        idyy= ihy*ii;
        String Nazvan= "strela_" + IntToStr(nom) + ".bmp";
        Strelka->LoadFromFile(Nazvan);
        Pysk_Strelu(idyy);
        nom++;
    }//for
} //Zapysk_Str1

void __fastcall TForm1::FormPaint(TObject *Sender)
{
    Mishen();
    Zapysk_Str1();
} //FormPaint

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    delete Fon;
    delete Strelka;
    Fon= NULL;
    Strelka= NULL;
} //FormDestroy

```

15.9.2. Применение ресурсов программы

В последнем приложении битовые образы загружались из файлов, которые необходимо поставлять пользователю приложения *вместе* с исполняемым файлом. Однако потребителям более удобно иметь дело *лишь с одним* исполняемым файлом, в котором *уже* находятся *все* необходимые для выполнения программы битовые образы. Битовые образы, расположенные *непосредственно* в исполняемом файле приложения, называются *ресурсом*. Операция копирования изображения из ресурса в динамический объект программы называется *загрузкой битового образа из ресурса*.

Ресурс создаётся при помощи специального файла, называемым *файлом ресурсов*. Ниже рассмотрены операции подключения ресурса к программе и загрузки из него битовых образов. Рассмотрим вначале последовательность действий, необходимую для порождения файла ресурсов. Этот процесс рассмотрим на примере прежнего учебного приложения и графического редактора *Builder* (утилиты) с именем *Image Editor*. Напоминаем, что он запускается командой *Tools/Image Editor* (или командой главного меню *Пуск/Программы/Borland C++ Builder 5 (или 6)/Image Editor*).



Рис. 15.19. Ввод команды *File/New/Resource File* в *Image Editor*

Последовательность действий по порождению файла ресурсов.

1. Введите команду горизонтального меню *Image Editor: File/New/Resource File* (см. рис. 15.19). В результате её выполнения создается пустой каталог ресурсов с именем по умолчанию *Untitled1*, а в горизонтальном меню редактора появится дополнительный пункт *Resource* (см. рис. 15.20).
2. С использованием пункта *Resource* введите команду *Resource/New/Bitmap*.
3. В появившемся окне (см. рис. 5.21, левая панель) с заголовком *Bitmap Properties* (параметры битового образа) введите габариты битового образа нашей программы (96x19) и установите режим максимальной цветности (256 цветов).
4. Нажмите кнопку *OK* окна *Bitmap Properties*, в результате создаётся первый (пока пустой) ресурс приложения с именем по умолчанию *Bitmap1*. Имя

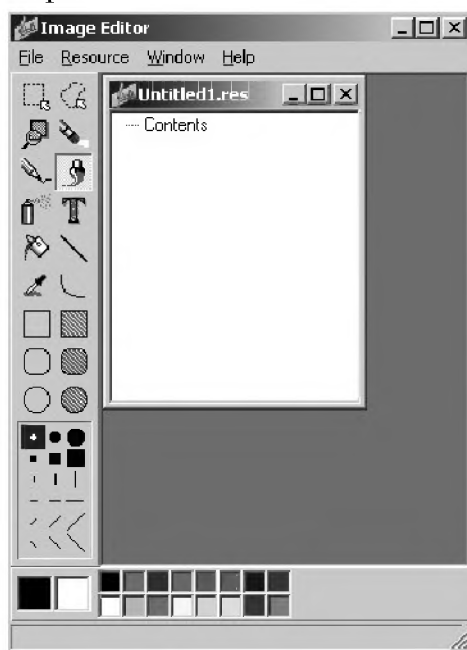


Рис. 15.20. Создан пустой файл ресурсов

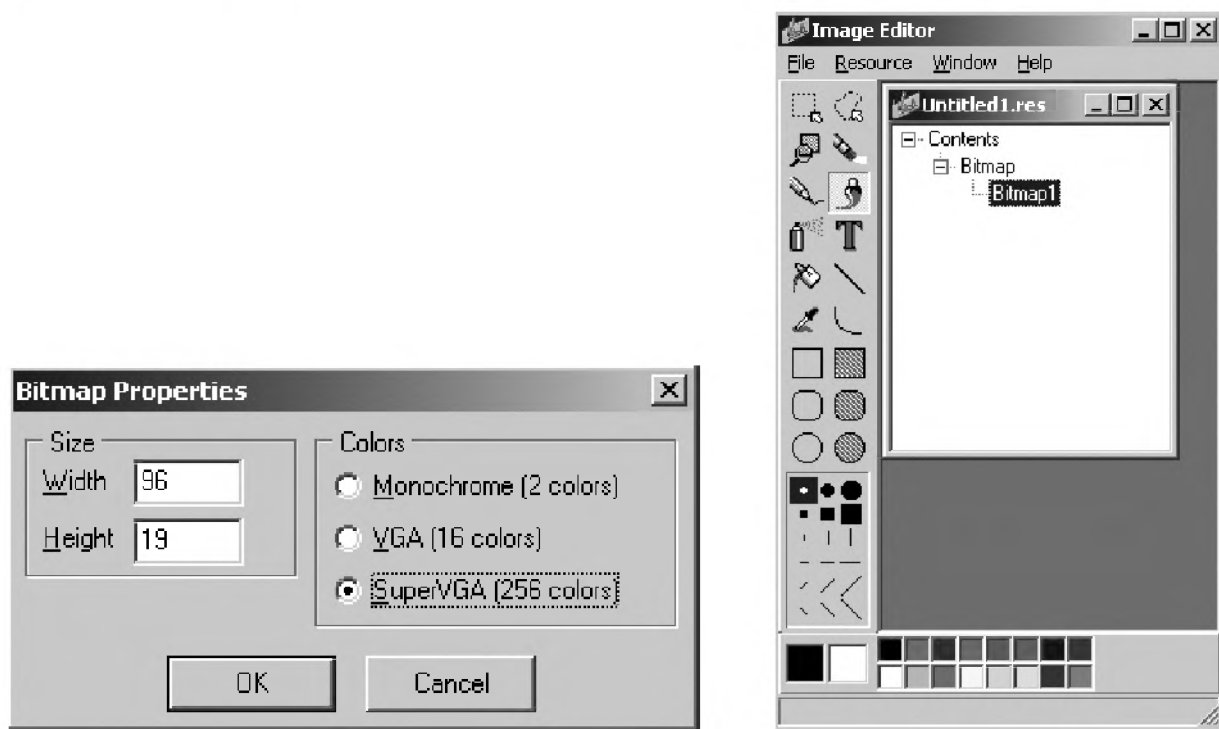


Рис. 15.21. Установка свойств первого ресурса (левая панель) и его появление в каталоге ресурсов (правая панель)

этого ресурса появится в каталоге *Bitmap* (вернее в дереве каталогов, см. рис. 5.21, правая панель). Далее, выделите ресурс *Bitmap1*, и при помощи команды *Resource/Rename* присвойте ему имя *FON*. Этот идентификатор можно набирать в любом регистре, после щелчка по свободной части окна все буквы окажутся прописными (они должны быть латинскими).

5. Команду *Resource/New/Bitmap* и прежние установки повторите ещё 9 раз для имён ресурсов *STRELA_1*, *STRELA_2*, ..., *STRELA_9*. В результате каталог ресурсов приобретёт вид показанный на рис. 15.22 (левая панель).
6. Примените команду *Resource/Edit* для *каждого* ресурса и с использованием инструментов редактора (подробнее см. п. 9.5.3) нарисуйте стелы разного цвета.
7. Однако для нашего приложения картинки уже имеются в файлах с одноимёнными именами и расширениями *bmp*. Поэтому проще скопировать их из этих файлов в ресурсы с соответствующими именами. С этой целью для каждого из 10 файлов следует повторить следующие операции.
 - 7.1. В редакторе *MS-Paint* открыть файл с картинкой, выделить всё изображение (*Ctrl+A*), после чего скопировать его в буфер обмена (*Ctrl+C*).
 - 7.2. С использованием команды *Edit/Paste* (*Ctrl+V*) горизонтального меню редактора *Image Editor* вставьте содержимое буфера в ресурс. Поскольку габариты битовых образов в файлах и пустых ресурсах *заблаговременно* установлены одинаковыми (96x19), поэтому битовые образы полностью заполнят полные рамки ресурсов после их вставки из буфера (см. рис. 15.22, правая панель).
8. По завершению работы над всеми ресурсами файл ресурсов следует сохранить (*Ctrl+S*) в том же каталоге, где размещены файлы разрабатываемого

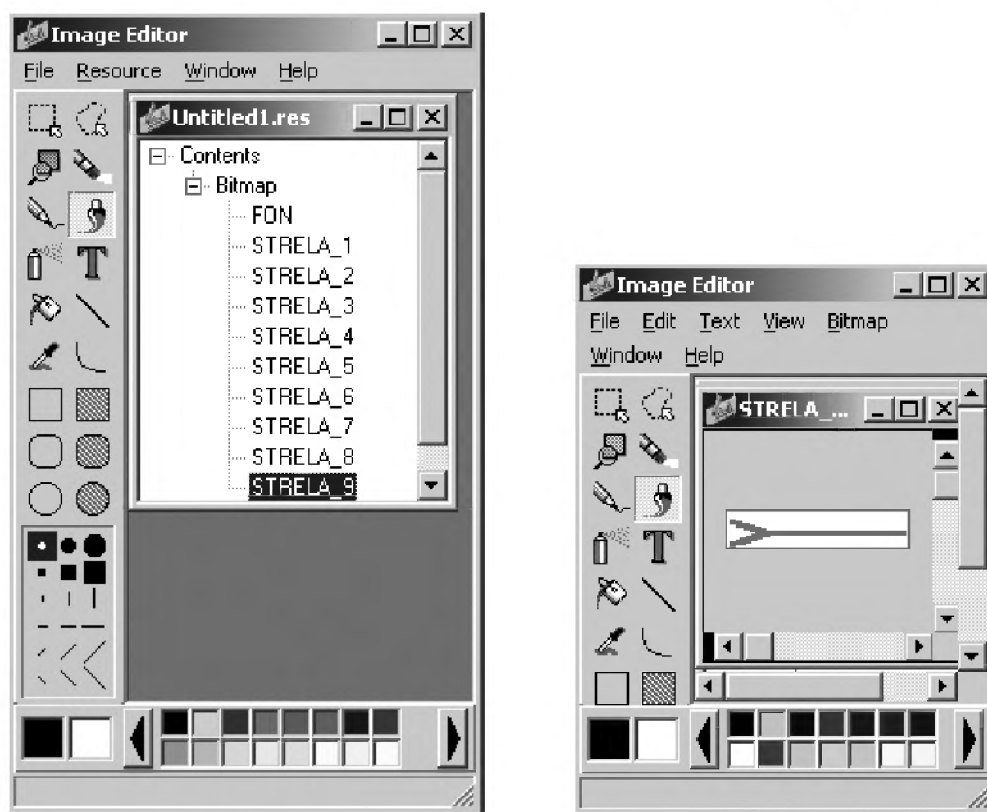


Рис. 15.22. Вид каталога после порождения всех ресурсов приложения (левая панель) и битовый образ ресурса с именем *STRELA_1* (правая панель)

приложения. Пусть, например, имя файла ресурсов будет *Fon_Strelu*. Редактор *Image Editor* файлу ресурсов присвоит расширение *res*.

Файл ресурсов создан. Для его подключения к исполняемому файлу проекта существует специальная директива компилятору. В нашем приложении эта директива выглядит вот так:

```
#pragma resource "Fon_Strelu.res"
```

Теперь после создания *полноценного* исполняемого файла (см. раздел 2.6) его можно передавать потребителям без файла ресурсов *Fon_Strelu.res*. Однако прежде познакомимся с устройством программы, в которой внедрён ресурс.

Загрузка битового образа из ресурса в динамический объект осуществляется методом *LoadFromResourceName*. В этом методе имеется два параметра: указатель исполняемого файла запущенной программы и имя её ресурса. Указатель запущенного исполняемого файла автоматически запоминается в глобальной переменной *HInstance*. Поэтому загрузка, например, фона в объект *Fon* осуществляется так:

```
Fon->LoadFromResourceName((int)HInstance,"FON");
```

В результате приведенных пояснений для вас должны быть ясными новые варианты функций *FormCreate* и *Zapysk_Strl*.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Form1->Align=alClient;//Интерфейс развёрнут на весь экран
    //Загрузка фона из ресурса
```

```
//Создаём объект для хранения графики
Fon= new Graphics::TBitmap();
//Загрузка изображение фона в Fon из ресурса
Fon->LoadFromResourceName((int)HInstance,"FON");
//Создаём ещё один объект для хранения графики
Strelka= new Graphics::TBitmap();
//Устанавливаем прозрачный задний план в будущем рисунке
Strelka->Transparent= true;
//Исходное положение левого конца стрелы
ix= -30;//Для достижения оси вертикальной мишени
} //FormCreate

void Zapysk_Strl(void)
{
    const int ihy= Y_Centr/6;
    for (int ii= -4, idyy, nom= 1; ii<= 4; ii++)
    {
        idyy= ihy*ii;
        String Nazvan= "STRELA_" + IntToStr(nom);
        Strelka->LoadFromResourceName((int)HInstance, Nazvan);
        Pysk_Strelu(idyy);
        nom++;
    } //for
} //Zapysk_Strl
```

Запустив это приложение, увидите, что каждая стрела оставляет след вдоль траектории своего полёта, аналогичный инверсионному следу самолёта. Почему так происходит? Настала очередь более подробно разобраться с цветами. В битовых образах нами выбрана максимальная палитра, состоящая из 256 цветов. Однако в перечислении *TColor* палитра значительно богаче (в ней имеется 256^3 цветов). Поэтому использованный нами для заднего плана мультфильма стандартный цвет окна (*clSilver*) не может быть полностью реализован даже максимальной палитрой битовых образов в 256 цветов (подбирается наиболее близкий цвет).

Что же следует предпринять для устранения этого эффекта? Можно выбрать цвет, который заведомо имеется в обеих палитрах. Например, это такие цвета, как белый, чёрный, красный, синий, зелёный и некоторые другие цвета (из-за большого числа цветов и цветовых оттенков палитры многих из них не имеют названия). Для заднего плана наиболее подходит белый цвет. Поэтому, если в начале тела функции *FormCreate* поставить строку:

```
Form1->Color= clWhite;
```

и загрузить этот цвет (прежним способом) в битовый образ объекта *FON*, то след от стирания стрел наблюдаться не будет.

Эту же команду можно реализовать несколько по-другому:

```
Form1->Color= 0x00FFFFFF;
```

Здесь цвет задаётся четырёхбайтовым шестнадцатеричным числом, три младших разряда которого определяют соответственно интенсивности красного, зелёного и синего цветов. Например, значение *0x00FF0000* соответствует чистому синему цвету, *0x0000FF00* – чистому зелёному, *0x000000FF* – чистому красному. Чёрный цвет описывается так: *0x00000000*.

Напоминаем, что основанием шестнадцатеричной системы счисления является число 16, помимо цифр от 0 до 9 (используемых в десятичной системе) в ней имеются цифры *A, B, C, D, E* и *F*, обозначающие соответственно числа 10, 11, 12, 13, 14 и 15. Цифры 16, как и цифры 10 в родной нам десятичной системе, там также нет. Для перевода двухразрядных чисел из шестнадцатеричной системы счисления в десятичную систему используется формула:

$$Chislo_{10} = q_1 \cdot 16 + q_0,$$

где q_1 и q_0 – цифры шестнадцатеричного числа первого и нулевого разрядов.

Если $q_1 = q_0 = F$, то $Chislo_{10} = 15 \cdot 16 + 15 = 255$. При $q_1 = q_0 = 0$ значение оказывается минимальным: $Chislo_{10} = 0$. Таким образом, интенсивность каждого чистого цвета имеет 256 значений, а суммарный цвет насчитывает 256^3 цветов и их оттенков.

Задавать интенсивности цветовых составляющих удобнее при помощи *MS-Word* (или *MS-Paint*, *MS-Excel*), где имеется возможность *визуально* подобрать нужный цвет, а затем в соответствующих окошках выяснить интенсивности его красной, зелёной и синей составляющих, поскольку там эти интенсивности выводятся в десятичной системе счисления. Перевод их к шестнадцатеричному виду удобно выполнить при помощи *MS-Excel* (или калькулятора из *Windows*), но можно и вручную, применяя, например, известную методику деления уголком.

Теперь установим назначение *старшего* байта. Обычно используется только три его значения:

- 00 – устанавливается ближайший цвет к заданному цвету из цветовой гаммы *системной* палитры;
- 01 – выбирается ближайший цвет из *текущей* палитры;
- 02 – берётся ближайший к заданному цвет из логической палитры активного в данный момент устройства.

В заключение напомним о том, что *после изменения* раскраски битовых образов в файле ресурсов, приложение следует откомпилировать не командой *Make* (*Ctrl+F9*), а командой *Build*. Это обусловлено тем обстоятельством, что в случае команды *Make* при подключении ресурса *Builder* не проверяет, изменялся ли ресурс или нет, – он подключает его к исполняемому файлу в режиме компиляции *Make* лишь один раз. Поэтому необходима *принудительная* компиляция *всех* частей создаваемого приложения командой *Build*.

15.10. Разработка мультфильма Аквариум

Основные графические возможности *C++Builder* закрепим в ходе разработки мультфильма *Аквариум*. Интерфейс приложения показан на рис. 15.23.

В этом приложении познакомимся с ещё одним событием формы. После автоматического порождения (событие *OnCreate*) и активизации видимости формы (событие *OnShow*) ей передаётся фокус управления (событие *OnActivate*). Функция *FormActivate*, предназначенная для обработки последнего события (она приведена ниже), разворачивает форму (её имя – *Akvarium*) на весь экран, определяет значение глобальных переменных *Shirina* и *Vusota*, описывающих

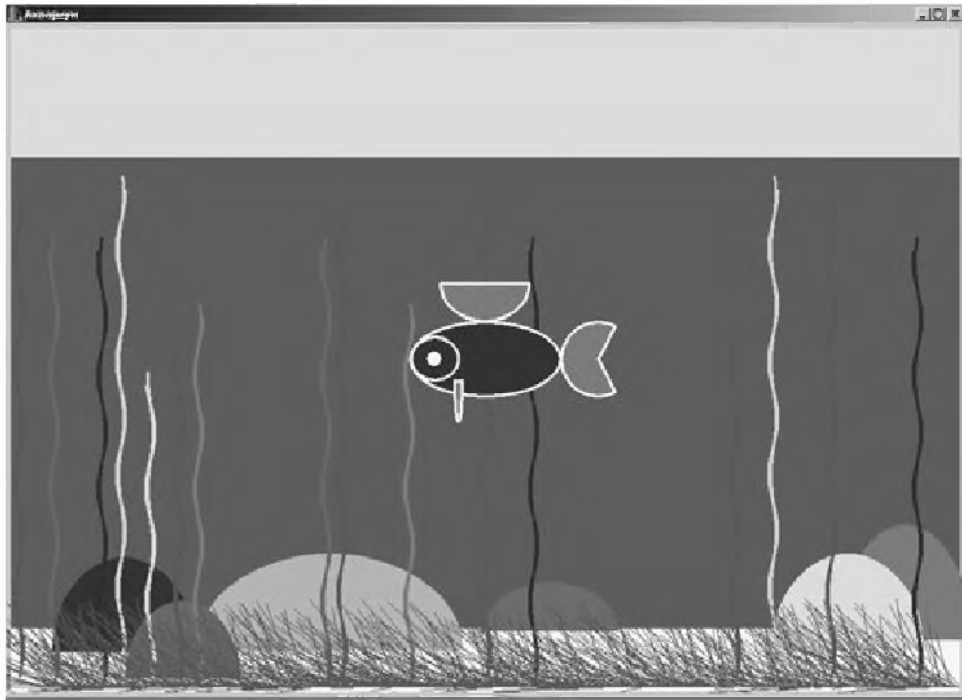


Рис. 15.23. Интерфейс приложения Аквариум

соответственно ширину и высоту порождённой формы.

Далее эта функция при помощи вызова пользовательской функции *Vozdyx_Voda_Pesok()* выводит соответствующими цветами прямоугольные области, изображающие воздух, воду и песок в аквариуме. Размеры и местоположения упомянутых прямоугольных областей выбраны такими, чтобы по периметру формы оставался прямоугольный контур серого цвета (цвет формы по умолчанию) толщиной в 5 пикселей. Этот серый контур определяет границы сосуда аквариума.

Затем пользовательские функции *Kamni()*, *Travka()* и *Vodorosli()* выводят разноцветные камни и траву на дне аквариума, извивающиеся водоросли различной длины и цвета.

Функция *Dvigen_Rubki()* осуществляет плавание рыбки в аквариуме. Движение рыбки реализовано первым способом мультипликации (три других способа также применимы). Ниже приводятся глобальные переменные и константы приложения, а также код функции *FormActivate*. Замети, что её тело можно переместить в функцию *FormPaint*. При этом работа программы не изменится. Функция *FormActivate* была выбрана лишь для знакомства с процессами, которые обеспечивают появление формы (они кратко описаны выше).

```
int Shirina, Vusota,
    iz= 1, //При iz== 1 рыбка движется влево
//Шаги перемещения рыбки по горизонтали и вертикали
    hx_Rub= 20, hy_Rub= -5;
//Радиусы эллипса (тела рыбки) по горизонтали и вертикали
const int iRx= 80, iRy= 40,
    Zadergka= 100; //Задержка изображения рыбки;

void __fastcall TАквариум::FormActivate(TObject *Sender)
{ //Интерфейс развёрнут на весь экран
  Аквариум->Align= alClient;
```

```

Shirina= Akvariym->ClientWidth,
Vusota= Akvariym->ClientHeight;
Vozdyx_Voda_Pesok();
Kamni();
Vodorosli();
Travka();
Dvigen_Rubki();
} //FormActivate

```

Сразу же за глобальными переменными и константами, впереди функции *FormActivate()*, следует расположить упомянутые ранее пользовательские функции. Функция *Vozdyx_Voda_Pesok()* может выглядеть, например, так:

```

void Vozdyx_Voda_Pesok(void)
{
    TRect Vozdyx, Voda, Pesok;
    //Воздух
    Akvariym->Canvas->Brush->Color= clAqua; //Голубой цвет
    Vozdyx= Rect(5, 5, Shirina - 5, Vusota/5);
    Akvariym->Canvas->FillRect(Vozdyx);
    //Вода, clTeal – цвет морской воды
    Akvariym->Canvas->Brush->Color= clTeal;
    Voda= Rect(5, Vusota/5, Shirina - 5, Vusota - 5);
    Akvariym->Canvas->FillRect(Voda);
    //Песок
    Akvariym->Canvas->Brush->Color= clYellow;
    Pesok= Rect(5, Vusota*0.9, Shirina - 5, Vusota - 5);
    Akvariym->Canvas->FillRect(Pesok);
} //Vozdyx_Voda_Pesok

```

Местоположение, форма и цвет каждого камня определяются функцией *Kamenj*, использующей метод *Pie* (рисует сектор). Формальные параметры заголовка этой функции: *ix* и *iy* – координаты центра сектора, *irx* и *iry* – его радиусы по горизонтали и вертикали, параметр *Cvet* – задаёт цвет заливки сектора-камня.

```

void Kamenj(int ix, int iy, int irx, int iry, TColor Cvet)
{
    Akvariym->Canvas->Pen->Color= Cvet;
    Akvariym->Canvas->Brush->Color= Cvet;
    Akvariym->Canvas->Pie(ix, Vusota-iy-2*iry, ix+2*irx, Vusota-iy,
                        ix+2*irx, Vusota-iy-iry, ix, Vusota-iy-iry);
} //Kamenj

```

Функция *Kamni()* вызовами функции *Kamenj()* строит целую серию камней с разными размерами, формой, цветом и местоположением (на песке в аквариуме).

```

void Kamni(void)
{
    Kamenj(Shirina*0.05, -50, 75, 98, clBlue);
    Kamenj(Shirina/5, -50, 140, 100, clSilver);
    Kamenj(Shirina/8, -60, 60, 80, clOlive);
    Kamenj(Shirina*0.87, -60, 70, 120, clFuchsia);
    Kamenj(Shirina*0.8, -50, 80, 100, clAqua);
    Kamenj(Shirina/2, 20, 70, 50, clGray);
} //Kamni

```

После «разбрасывания» камней выводятся водоросли. Каждая водоросль рисуется функцией *Vodorosl(int ix, int iy, int iDlina, TColor Cvet)*. Здесь *ix, iy* – координаты корня водоросли, её самой нижней точки; параметры *iDlina, Cvet* соответственно задают её длину и цвет. Максимальная ширина стебля водоросли составляет 6 пикселей и изображается при помощи 6 синусоид. Три первые синусоиды синфазны, имеют максимальную длину, амплитуда каждой из них равна 5, абсциссы их центральных осей смещены друг относительно друга по горизонтали на 1 пиксель (для утолщения водоросли). Три последующие синусоиды также синфазны между собой и каждая из них смещена относительно соседней синусоиды по горизонтали на 1 пиксель (для достижения прежней цели), однако их фазы увеличены на 0,8 радиана относительно первой тройки синусоид, они имеют в два раза меньшую амплитуду (2,5) и уменьшенную в 0,98 раз длину. Этим обеспечивается более естественный вид водоросли: она имеет переменную толщину, её верхушка не является плоской.

```
void Vodorosl(int ix, int iy, int iDlina, TColor Cvet)
{
    int ix1, ix2, iy1, iy2;
    for (int iL= 1; iL< iDlina; iL++)
    {
        ix1= ix+5*sin(3*iL*M_PI/180);
        ix2= ix+2.5*sin(3*iL*M_PI/180+0.8);
        iy1= Vusota-iy-iL;
        iy2= (Vusota-iy)*0.98-iL;
        Akvariym->Canvas->Pixels[ix1][iy1]= Cvet;
        Akvariym->Canvas->Pixels[ix1+1][iy1]= Cvet;
        Akvariym->Canvas->Pixels[ix1+2][iy1]= Cvet;
        Akvariym->Canvas->Pixels[ix2][iy2]= Cvet;
        Akvariym->Canvas->Pixels[ix2+1][iy2]= Cvet;
        Akvariym->Canvas->Pixels[ix2+2][iy2]= Cvet;
    } //for
} //Vodorosl
```

Функция *Vodorosli()* рассаживает водоросли разной длины по всему аквариуму и задаёт их цвет.

```
void Vodorosli(void)
{
    Vodorosl(Shirina/2, 20, Vusota*0.7, clGreen);
    Vodorosl(Shirina*0.013, 20, Vusota*0.7, clGreen);
    Vodorosl(Shirina*0.86, 20, Vusota*0.7, clGreen);
    Vodorosl(Shirina*0.76, 20, Vusota*0.49, clGreen);
    Vodorosl(Shirina*0.18, 20, Vusota*0.49, clGreen);
    Vodorosl(Shirina*0.35, 20, Vusota*0.67, clGreen);
    Vodorosl(Shirina*0.35, 20, Vusota*0.57, clGreen);
    Vodorosl(Shirina*0.55, 10, Vusota*0.65, clMaroon);
    Vodorosl(Shirina*0.95, 10, Vusota*0.65, clMaroon);
    Vodorosl(Shirina/3, 10, Vusota*0.65, clOlive);
    Vodorosl(Shirina*0.05, 12, Vusota*0.65, clOlive);
    Vodorosl(Shirina*0.8, 25, Vusota*0.72, clLime);
    Vodorosl(Shirina*0.12, 25, Vusota*0.72, clLime);
    Vodorosl(Shirina*0.15, 10, Vusota*0.45, clAqua);
}
```



```
Vodorosl(Shirina*0.2, 10, Vusota*0.55, clFuchsia);
Vodorosl(Shirina*0.42, 10, Vusota*0.55, clFuchsia);
Vodorosl(Shirina*0.1, 10, Vusota*0.65, clMaroon);
} // Vodorosli
```

Трава изображается функцией *Travka()* в виде отрезков прямой толщиной в один пиксель, длина и наклон отрезков изменяются по случайному закону в заданных интервалах. Различие длин травинок не превышает 99 пикселей (объясните, почему не 100), «посажены» травинки так, чтобы они не выходили за границы сосуда – серого прямоугольного контура.

```
void Travka(void)
{
    Akvariym->Canvas->Pen->Width= 1;
    Akvariym->Canvas->Pen->Color= clGreen;
    for (int ii= 40; ii< Shirina - 5; ii++)
    {
        Akvariym->Canvas->MoveTo(ii, Vusota - 10);
        Akvariym->Canvas->LineTo(ii - 10 - random(100),
                                Vusota - 5 - random(100));
    } //for
} //Travka
```

Функция *Rubka* при помощи различных примитивов конструирует изображение рыбки. Её аргументами являются: *ix, iy* – координаты центра тела рыбки, эллипса с радиусами *iRx, iRy* по горизонтали и вертикали; *Cvet_Kont* – цвет контура всех примитивов, *Cvet_Tela* – цвет тела, *Cvet_Xv* – цвет хвоста (и всех плавников), *Cvet_Gl* – цвет глаза.

Параметры головы вместе с глазом (эллипсы), плавников и хвоста (секторы) определяются линейными зависимостями *iRx* и *iRy*. Поэтому при изменении размеров тела рыбки (параметров *iRx* и *iRy*) все другие составляющие её примитивы изменяются пропорционально и не меняют своего относительного взаиморасположения. В результате при увеличении или уменьшении *iRx* и *iRy* рыбка не разваливается и не искажается, все её элементы остаются сгруппированными.

В ходе перемещения рыбки в аквариуме (напоминаем, оно осуществляется функцией *Dvigen_Rubki*) всякий раз по достижению рыбой поверхности воды, камней, вертикальных стенок сосуда изменяется направление её движения на противоположное. Это оказывается возможным при помощи функции *Dvigen_Rubki*, в которой, например, в случае вертикальных стенок аквариума происходит изменение знака глобальной переменной *iz* всякий раз, когда рыбка приближается до стенок на заданное расстояние (более подробно об этом сообщается при описании указанной функции). Упомянутое значение минимально допустимого приближения должно быть несколько большим горизонтального габарита хвоста. В противном случае стенки сосуда при повороте рыбки (вокруг центральной вертикальной оси её тела) будут разрушены (стёрты). При изменении направления движения по горизонтали у рыбки голова и хвост меняются местами, изменяется положение нижнего плавника. Это происходит в результате изменения знака переменной *iz*.

```

void Rubka(int ix, int iy, int iRx, int iRy, TColor Cvet_Kont, TColor
          Cvet_Tela, TColor Cvet_Xv, TColor Cvet_Gl)
{
    TRect Telo, Golova, Glaz, Verxn_Plavn, Xvost, V_Plav1, V_Plav2;
    Akvariym->Canvas->Pen->Color= Cvet_Kont;
    Akvariym->Canvas->Pen->Width= 3;

    //Тело
    Akvariym->Canvas->Brush->Color= Cvet_Tela;
    Telo= Rect(ix-iRx, iy-iRy, ix+iRx, iy+iRy);
    Akvariym->Canvas->Ellipse(Telo);

    //Голова
    Golova= Rect(ix-iz*iRx*0.98, iy-iRy*0.6, ix-iz*iRy*0.7, iy+iRy*0.6);
    Akvariym->Canvas->Ellipse(Golova);

    //Глаз
    Akvariym->Canvas->Brush->Color= Cvet_Gl;
    Glaz= Rect(ix-iz*iRx*0.76, iy-iRy*0.17,
              ix-iz*iRx*0.6, iy+iRy*0.17);
    Akvariym->Canvas->Ellipse(Glaz);

    //Верхний плавник
    Akvariym->Canvas->Brush->Color= Cvet_Xv;
    Akvariym->Canvas->Pie(ix-iRx*0.6, iy-iRy*3, ix+iRx*0.6, iy-iRy,
                       ix-iRx*0.6, iy-iRy*2, ix+iRx*0.6, iy-iRy*2);

    if (iz== 1)//Движение влево
    {
        //Хвост
        Akvariym->Canvas->Brush->Color= Cvet_Xv;
        Akvariym->Canvas->Pie(ix+iz*iRx*1, iy-iRy, ix+iz*iRx*2, iy+iRy,
                           ix+iz*iRx*2, iy-iRy*2, ix+iz*iRx*2, iy+iRy*2);

        //Нижний плавник
        Akvariym->Canvas->Brush->Color= Cvet_Xv;
        Akvariym->Canvas->Pie(ix-iz*iRx*0.4, iy-iRy*0.5, ix-iz*iRx*0.3,
                           iy+iRy*1.7, ix-iz*iRx*0.4, iy+iRy*0.6, ix-iz*iRx*0.3, iy+iRy*0.6);
    }//if_1
    if (iz== -1)//Движение вправо
    {
        //Хвост
        Akvariym->Canvas->Brush->Color= Cvet_Xv;
        Akvariym->Canvas->Pie(ix+iz*iRx*1, iy-iRy, ix+iz*iRx*2, iy+iRy,
                           ix+iz*iRx*2, iy+iRy*2, ix+iz*iRx*2, iy-iRy*2 );

        //Нижний плавник
        Akvariym->Canvas->Brush->Color= Cvet_Xv;
        Akvariym->Canvas->Pie(ix-iz*iRx*0.4, iy-iRy*0.5, ix-iz*iRx*0.3,
                           iy+iRy*1.7, ix-iz*iRx*0.3, iy+iRy*0.6,
                           ix-iz*iRx*0.4, iy+iRy*0.6);
    }//if_2
} //Rubka

```

Вначале функция *Dvigen_Rubki* выводит изображение рыбки по центру формы ($ix = Shirina/2$, $iy = Vusota/2$), затем происходит перемещение рыбки (влево и

вверх) с заданными шагами по горизонтали и вертикали. В ходе работы приложения направление поступательного движения рыбки изменяется всякий раз при наступлении определённых событий. Поступательное перемещение в неизменном направлении происходит до тех пор, пока габаритные размеры рыбки не достигнут вертикальных стенок (с учётом упомянутого эффекта при развороте), глубины расположения верхних частей камней или поверхности воды. В противном случае изменяется знак соответствующего шага перемещения (*hx_Rub* или *hy_Rub*).

Рыбка плавает до четвёртого разворота, далее её изображение пропадает. После очередного поворота рыбки, съеденные ею водоросли (стираемые при скрывании рыбки на каждом шаге её перемещения), восстанавливаются при помощи повторного вывода функции *Vodorosli()*.

```
void Dvigen_Rubki(void)
{
    int ix= Shirina/2, iy= Vusota/2, ik= 0;
    do
    {
        if ((ix< iRx*2+20)|| (ix> Shirina-iRx*2-20))
        {
            ik++; //Счётчик разворотов рыбки
            iz= -iz;
            hx_Rub= -hx_Rub;
            Vodorosli(); //Восстановление съеденных водорослей
        } //if_1
        if ((iy< Vusota/5+iRy*2 + abs(hy_Rub) + 10)||
            (iy> Vusota-iRy*2-abs(hy_Rub)-200))
        {
            hy_Rub= -hy_Rub;
            Vodorosli(); //Восстановление съеденных водорослей
        } //if_2
        Rubka(ix, iy, iRx, iRy, clWhite, clBlue, clFuchsia, clYellow);
        Sleep(Zadergka); //Задержка изображения рыбки
        //Стираем изображение рыбки первым способом
        Akvariym->Canvas->Brush->Color= clTeal;
        Rubka(ix, iy, iRx, iRy, clTeal, clTeal, clTeal, clTeal);
        ix-= hx_Rub;
        iy+= hy_Rub;
    } //do
    while (ik<4);
} //Dvigen_Rubki
```

Функция *FormClick* (обработчик события *OnClick*) осуществляет выход из программы после щелчка мышью по форме. Этот щелчок можно осуществить, не дожидаясь последнего поворота рыбки: рыбка закончит свою задачу, а затем уж будет закрыто приложение.

```
void __fastcall TAKvariym::FormClick(TObject *Sender)
{
    exit(0);
} //FormClick
```

15.11. Вариант мультфильма *Аквариум*, разработанный на основе применения пользовательских классов

В предшествующем разделе реализован *процедурный* подход к разработке приложений. Полагаем, что для решения *обсуждаемой* учебной задачи он наиболее оптимален. Однако следует потренироваться реализовывать приёмы ООП, эффективность которых для известных условий (см. четырнадцатый урок) неоспорима. Поэтому покажем, как приложение *Аквариум* может выглядеть в случае использования классов.

Оставим без изменения все глобальные переменные и константы прежнего приложения, однако большинство пользовательских функций заменим методами соответствующих классов. Классы, как матрёшки вложим друг в друга. Пусть самым внутренним родительским классом будет *TVozdyx*.

```
class TVozdyx
{
    protected:
        TRect Vozd;
    public:
        void Inic(void);
        void Vuvod(void);
}; //TVozdyx
```

Для удобства изучения приложения все классы опишем не в заголовочном файле (*Ctrl_F6*), а в файле реализации, непосредственно *перед* кодом их методов. Поэтому сразу же после описания класса *TVozdyx* в файле реализации нового приложения приведём коды методов инициализации и вывода прямоугольной области, изображающей воздух аквариума.

```
void TVozdyx::Inic(void)
{
    Vozd= Rect(5, 5, Shirina-5, Vusota/5);
} //TVozdyx::Inic

void TVozdyx::Vuvod(void)
{
    Akvariym->Canvas->Brush->Color= clAqua;
    Akvariym->Canvas->FillRect(Vozd);
} //TVozdyx::Vuvod
```

В классе *TVoda* осуществим открытое наследование класса *TVozdyx* и переопределим все его методы. Метод *Vuvod()* теперь выводит обе прямоугольные области – воздух и воду.

```
class TVoda: public TVozdyx
{
    protected:
        TRect Vod;
    public:
        void Inic(void);
```

```
class Takvar_Kamni: public TPesok
{
    protected:
        void Kamenj(int ix, int iy, int irx, int iry, TColor Cvet);
        void Kamni(void);
    public:
        void Vuvod(void);
}
```

```

}; //TAKvar_Kamni

void TAKvar_Kamni::Vuvod(void)
{
    TPesok::Vuvod();
    Kameni();
}; //TAKvar_Kamni::Vuvod

void TAKvar_Kamni::Kamenj(int ix, int iy, int irx, int iry, TColor Cvet)
{ //тело метода совпадает с телом функции Kamenj()
}; //TAKvar_Kamni::Kamenj

void TAKvar_Kamni::Kamni(void)
{ //тело метода совпадает с телом функции Kamni()
}; //TAKvar_Kamni::Kamni

```

Класс *TAKvar_Kamni_Vodorosli* является наследником родительского класса *TAKvar_Kamni*, поэтому, как следует из его названия, помимо прежних элементов аквариума он выводит и все водоросли.

```

class TAKvar_Kamni_Vodorosli: public TAKvar_Kamni
{
    protected:
        void Vodorosl(int ix, int iy, int iDlina, TColor Cvet);
        void Vodorosli(void);
    public:
        void Vuvod(void);
}; //TAKvar_Kamni_Vodorosli

void TAKvar_Kamni_Vodorosli::Vodorosl(int ix, int iy,
                                       int iDlina, TColor Cvet)
{ //тело метода совпадает с телом функции Vodorosl()
}; //TAKvar_Kamni_Vodorosli::Vodorosl

void TAKvar_Kamni_Vodorosli::Vodorosli(void)
{ тело метода совпадает с телом функции Vodorosli()
}; // TAKvar_Kamni_Vodorosli::Vodorosli

void TAKvar_Kamni_Vodorosli::Vuvod(void)
{
    TAKvar_Kamni::Vuvod();
    Vodorosli();
}; //TAKvar_Kamni_Vodorosli::Vuvod

```

Класс *TAKvar_Kam_Vod_Trava* помимо прежних элементов аквариума выводит траву на его песчаном дне.

```

class TAKvar_Kam_Vod_Trava: public TAKvar_Kamni_Vodorosli
{
    public:
        void Vuvod(void);
}; //TAKvar_Kam_Vod_Trava

void TAKvar_Kam_Vod_Trava::Vuvod(void)
{
    TAKvar_Kamni_Vodorosli::Vuvod();
    Akvariym->Canvas->Pen->Width= 1;
    Akvariym->Canvas->Pen->Color= clGreen;
}

```

Последним дочерним классом иерархии является класс *TAkvar_Kam_Vod_Rub*. Он позволяет в полной мере реализовать все эффекты прежнего приложения *Аквариум*.

Теперь функция *FormActivate* выглядит так:

Функция *FormClick* в настоящем приложении осталась прежней.

- ❑ Назовите типы визуальных объектов, на поверхность которых можно выводить графические примитивы. Какое свойство этих объектов при этом используется?
- ❑ Назовите свойство визуальных объектов типа *TImage*, предназначенное для хранения изображений графических файлов? Какого типа могут быть

эти файлы? Назовите основные отличительные черты хранения графической информации в таких файлах.

- ☐ С какой целью применяется свойство *Transparent*?
- ☐ Как на поверхности объекта типа *TImage* осуществить просмотр графических файлов с расширением *jpg*?
- ☐ Где находится точка канвы с максимальными координатами по горизонтали и вертикали?
- ☐ Перечислите методы канвы, в которых аргумент может быть структурного типа или только структурного типа.
- ☐ Укажите свойство канвы, которое представляет собой двумерный массив.
- ☐ Какой метод канвы цветом кисти рисует только контур прямоугольника.
- ☐ Перечислите три способа задания полей структуры, необходимой для вывода прямоугольника.
- ☐ Перечислите способы вывода текста на поверхности канвы.
- ☐ Что произойдёт, когда в методе *Pie* последние две пары его аргументов поменять местами?
- ☐ Назовите стиль кисти, обеспечивающий прозрачность заднего плана выводимого текста.
- ☐ Как программно установить требуемый стиль шрифта для текста, выводимого на канве?
- ☐ Что собой представляет переменная типа *TColor*?
- ☐ Разъясните сущность битовых образов и ресурса приложения.
- ☐ Назовите метод канвы с полиморфным параметром.
- ☐ Укажите назначение метода *Draw* и всех его параметров?
- ☐ Почему в программах явно не применяется метод *Free*?
- ☐ Назовите директиву компилятору, включающую файл ресурсов в исполняемый файл.

15.12. Задачи для программирования

Задача 15.1. Модернизируйте приложение *Пособие* таким образом, чтобы имелась возможность сохранять на интерфейсе изображений кривых с выбранными параметрами. Для таких кривых используйте серый цвет.

Задача 15.2. На основе приложения *Полёт стрел* (см. раздел 15.8, первый способ мультипликации) разработайте программу, в которой стрела пролетала бы сквозь мишень и оставляла в ней отверстие, совпадающее по цвету с цветом стрелы.

Задача 15.3. На основе приложения *Полёт стрел* разработайте программу, в которой стрела пролетала бы сквозь мишень, и оставляла в ней отверстие красного цвета. Перед вылетом очередной стрелы отверстие от предшествующей стрелы

должно исчезать. Для восстановления изображения мишени (испорченного пролётом стрелы) используйте не функцию *Mishen*, а метод *CopyRect*.

15.13. Варианты решения задач

Решение задачи 15.1. На интерфейсе приложения *Пособие* следует добавить кнопку с заголовком *Сохранить* и именем, например, *Soxranitj*. Код функции по обработке нажатия этой кнопки показан ниже.

```
void __fastcall TForm1::SoxranitjClick(TObject *Sender)
{
    //Прежние графики выводятся серым цветом
    Cvet_Par= clGray;
    Cvet_Okr= clGray;
    TForm1::FormPaint(Form1);
} //SoxranitjClick
```

Решение задачи 15.2.

Для решения этой задачи в приложении *Полёт стрел* функцию *Pysk_Strelu* следует заменить таким кодом:

```
void Pysk_Strelu(int idy, TColor Cvet_Strelu)
{
    const int dl_Strelu= Form1->ClientWidth*0.1; //Длина
    for (int ii= 0, ix; ii< 8; ii++)
    {
        ix= dl_Strelu + ii*dl_Strelu;
        Form1->Canvas->Pen->Color= Cvet_Strelu;
        Strela(ix,idy);
        Sleep(200);
        Form1->Canvas->Pen->Color= clSilver;
        Strela(ix,idy);
        if (ii== 7)
        {
            //Для заделки прорех в мишени после пролёта стрелы
            Mishen();
            Form1->Canvas->Brush->Color= Cvet_Strelu;
            Form1->Canvas->Ellipse(ix+dl_Strelu-3, Y_Centr+idy-5,
                                   ix+dl_Strelu+3, Y_Centr+idy+5);
        }
    } // if
} //for
} //Pysk_Strelu
```

Решение задачи 15.3.

Для решения этой задачи в приложении *Полёт стрел* функцию *Pysk_Strelu* следует заменить следующим кодом:

```
void Pysk_Strelu(int idy)
{
    while (ix< Centr_Mish-rx*n+w/2 ) //Полёт до мишени
    {
        Strel_Rect= Bounds(0, 0, Form1->ClientWidth, Form1->ClientHeight);
        Fon_Rect= Bounds(0, 0, Form1->ClientWidth, Form1->ClientHeight);
        //Рисуем стрелу стандартным методом Draw
```

```
Form1->Canvas->Draw(ix, Y_Centr+idy, Strelka);
Sleep(100); //Задержка выполнения программы
//Стираем стрелу при помощи объекта Fon
Form1->Canvas->CopyRect(Strel_Rect, Fon->Canvas, Fon_Rect);
ix += w; //w- шаг перемещения стрелы
if (ix > Centr_Mish-rx*n+w/2) //Стрела пронзила мишень
{
    Form1->Canvas->Brush->Color= clRed;
    Form1->Canvas->Ellipse(Centr_Mish-3,
        Y_Centr+idy+h/2-5, Centr_Mish+3, Y_Centr+h/2+idy+5);
    Sleep(500);
} //if
} //while
ix= -30; //Возврат аргумента в исходное состояние
} //Pysk_Strelu
```



Урок 16. Запуск чужих программ из вашего приложения

16.1. Технология OLE

Использование терминов имеет первостепенное значение для лаконичного изложения информации в *любой* сфере человеческой деятельности. Невозможно обойтись без них и в компьютерных технологиях. Однако в виду бурного развития упомянутых технологий их терминология ещё окончательно не сложилась (имеются разные термины для одних и тех же понятий), что несколько *мешает* беглому пониманию отдельных стандартных выражений. Поэтому для ясности последующего повествования напомним смысл отдельных слов и словосочетаний лексикона операционной системы *Windows* и её приложений.

Компьютерная программа, применяемая для создания, например текста научного отчёта (картинки, таблицы и др.), называется обычно *приложением*. Продукт, полученный при помощи того или иного приложения, именуется *документом* (текст, картинка, таблица и др.).

В любой документ, скажем приложения *Word*, можно вставить, например, документ приложения *Excel* или *Paint* (либо их фрагменты). Таким образом, документы разных приложений можно *вставлять* друг в друга. Если требуется отредактировать (внести изменения) во вставленный документ (объект), то по нему выполняется двойной щелчок. В результате открываются команды и пиктограммы того приложения, в котором документ был изготовлен. Вставленный объект называется *внедрённым* в основной документ, он не связан с документом-источником. Поэтому изменения исходного документа (копия которого была вставлена) в основном документе отражаться *не будут*.


Однако вставку можно выполнить и таким образом, чтобы её редактирование в основном документе отражалось в исходном документе и наоборот: редактирование исходного документа проявлялось бы в основном документе. Такой метод вставки (вид внедрения) называется *связыванием*. В случае связывания в основной документ *программно* вставляется *только имя* файла вставляемого документа (фактически: указатель на документ, расположенный в файле). Ясно, что в этом случае потребителю следует передавать *оба* файла: с основным документом и вставленным объектом. Упомянутый метод вставки удобен, например, в случае, когда заданный объект внедряется *во многие* документы и в будущем *планируется* его редактирование.

Все стандартные приложения *Windows* имеют возможность взаимно внедрять

и связывать свои документы. При этом программы, при помощи которых разработаны эти документы, могут быть выпущены как одной и той же, так и разными фирмами. Технология таких вставок в *Windows* называется технологией *OLE: Object Linking and Embedding* (связывание и внедрение объекта). Теперь пришло время вооружить и ваши приложения мощным инструментарием этой технологии.

16.1.1. Знакомство на примере программы

Ваша программа должна получить способность запускать выбранное *чужое* приложение для разработки его *нового* документа. В программе этот документ может быть как внедрённым, так и связанным. Следует предусмотреть средства для редактирования такого документа *при повторном* его *открытии*, а также обеспечить возможность *закрытия* документа. При этом язык программирования, применённый для написания чужого приложения (запускаемого вашей программой), может быть произвольным. Для выполнения таких и аналогичных операций исторически (хронологически) первой в *C++Builder* стала использоваться технология *OLE*, одноимённая с описанной выше технологией *Windows*. Рассмотрим её применение вначале на примере, реализующем перечисленные выше задачи. Для разработки учебного приложения повторяйте упомянутые ниже действия.

На форме с заголовком *Пример OLE* расположите объект *OleContainer1* (закладка *System*). Этот компонент обеспечивает внедрение и связывание, на *Палитре компонентов* он выглядит вот так: . После порождения объекта он должен занимать всю свободную часть формы. Для этого в *Инспекторе Объектов* его свойству *Align* выберите значение *alClient*. Этот же эффект достигается программно, если, например, в конструкторе приложения включить такую строку: `OleContainer1->Align= alClient;`

Далее на форме (внутри контейнера *OleContainer1*) разместите объекты: главное меню *MainMenu1* (закладка *Standard*) и диалоги *OpenDialog1* и *SaveDialog1* (закладка *Dialogs*).

В *MainMenu1* введите пункт с заголовком *Объект* (его имя *Object*) и его подпункты с заголовками: *Новый*, *Открыть*, *Сохранить* и *Закрыть*. Имена этих подпунктов пусть будут соответственно такими: *PNew*, *POpen*, *PSave*, *PClose* (*P* – от слова *пункт*).

При помощи *Инспектора Объектов* (страница *Properties*) в диалогах *OpenDialog1* и *SaveDialog1* в окне ввода свойства *Filter* впишите текст «Объекты OLE[*.*|Все файлы]*.*». Более наглядно эта же операция выполняется с использованием визуального редактора *Filter Editor*. Этот редактор, как и все другие визуальные редакторы, вызывается нажатием пиктограммы с тремя точками. В нашем случае она расположена справа от окна ввода свойства *Filter*. В двух строках и колонках появившейся таблицы (см. рис. 16.1) наберите текст: «Объекты OLE», «*.*», «Все файлы» и «*.*». В свойстве *DefaultExt* всех диалогов укажите расширение *ole*, теперь это расширение будет выбираться по умолчанию.

Рис. 16.1. Установка параметров свойства *Filter*

Перечисленные операции можно, конечно же, осуществить и при помощи программного кода, например, в конструкторе приложения. В окончательном виде конструктор может выглядеть так:

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    OleContainer1->Align=alClient;
    OpenFileDialog1->DefaultExt= "ole";
    OpenFileDialog1->Filter="Объекты OLE|*.ole|Все файлы| *.*";
    SaveDialog1->DefaultExt= "ole";
    SaveDialog1->Filter="Объекты OLE|*.ole|Все файлы| *.*";
} //TForm1
```

В описании класса формы в разделе *public* объявите глобальную общедоступную переменную строкового типа, она потребуется в частности для запоминания текущего имени файла, предназначенного для хранения объекта *OLE*:

```
String FName; //Имя открываемого файла
```

Функцию *PNewClick* (для обработки события, вызванного нажатием подпункта *Новый*) сделайте такой:

```
void __fastcall TForm1::PNewClick(TObject *Sender)
{
    if (OleContainer1->InsertObjectDialog())
    {
        FName= "";
        OleContainer1->DoVerb(ovShow);
    } //if
} //PNewClick
```

Метод *InsertObjectDialog()* класса *TOleContainer* осуществляет обращение к стандартному окну *Windows* с именем *Вставка объекта* (в нерусифицированной *Windows* – *Insert Object*). При помощи этого окна пользователь выбирает тип вставляемого объекта, порождает его или открывает уже существующий объект заданного типа. Далее функция *PNewClick* загружает этот объект в контейнер

OleContainer1. При этом происходит очистка переменной *FName*. Она необходима в случае, когда перед выбором пункта *Новый* вызывался (активизировался) пункт *Открыть* или *Сохранить* (назначение этих пунктов и коды функций для их обработки приведены ниже). Если очистку не производить, то сохранение создаваемого объекта будет предложено выполнить под именем, которое осталось в этой переменной после активизации одного из упомянутых пунктов (что, в принципе, допустимо, однако не всегда удобно).

Метод *DoVerb* с аргументом *ovShow* (показать объект) обеспечивает *немедленное* открытие программы, используемой при порождении вставленного или вставляемого объекта. Взамен значения *ovShow* можно применить также стандартное значение *ovPrimary* (основное назначение), что позволяет осуществить действия, которые запрограммированы для *OLE*-объекта по умолчанию. Обычно это активация объекта, поэтому замена *ovShow* на *ovPrimary* не изменит работу программы.

Обработку события, вызванного нажатием подпункта *Открыть*, осуществляет функция *POpenClick*:

```
void __fastcall TForm1::POpenClick(TObject *Sender)
{
    if (OpenDialog1->Execute())
    {
        OleContainer1->LoadFromFile(OpenDialog1->FileName);
        FName= OpenDialog1->FileName;
        OleContainer1->DoVerb(ovShow);
    } //if
} //POpenClick
```

Эта функция вызывает стандартное диалоговое окно, в котором пользователь выбирает желаемый файл *с уже существующим OLE-объектом* (но не файл какого-то продукта *с иным расширением*). Поэтому указанным пунктом можно воспользоваться только, в том случае, если *OLE-файл ранее был создан*. Далее упомянутый *OLE-объект* загружается в контейнер *OleContainer1* методом *LoadFromFile*. Имя выбранного в диалоге файла запоминается в глобальной переменной *FName*.

Последняя строка тела оператора *if* этой функции методом *DoVerb* (также как и в функции *PNewClick*) открывает программу, связанную с загруженным объектом (согласно расширению исходного файла, который позже был преобразован в *OLE-объект* (файл)). Ещё раз напоминаем, что разделом *Открыть* следует пользоваться *только* для открытия *OLE-файлов*. Поэтому согласно настройке свойства *DefaultExt* по умолчанию пользователю программы будет предложено расширение *ole*, однако у него *имеется* возможность, помимо допустимого объекта (*Объекты OLE*), выбрать и файл с произвольным расширением (*Все файлы*).

Мы умышленно не ограничили возможности фильтра для того, чтобы познакомить вас с обработкой исключительной ситуации, которая возникнет при выборе файла с иным расширением (не *ole*). В упомянутой функции предусмотрим теперь обработку исключительной ситуации, обусловленной неправильным вызовом метода *LoadFromFile* класса *TOleContainer*. Модернизированную функции

POpenClick приводим ниже.

```
void __fastcall TForm1::POpenClick(TObject *Sender)
{
    if (OpenDialog1->Execute())
        if (OpenDialog1->Options.Contains(ofExtensionDifferent))
        {
            ShowMessage("Разрешён выбор файлов только с\
                               расширением ole ");
            Abort();
        } //if_2
    else
    {
        OleContainer1->LoadFromFile(OpenDialog1->FileName);
        FName= OpenDialog1->FileName;
        OleContainer1->DoVerb(ovShow);
    } //else
} //POpenClick
```

Значения свойства *Options* являются элементами множества, которое по умолчанию является пустым множеством. Если при помощи *Инспектора Объектов* осуществляется назначение какой-либо константе множества значения *true*, то при этом происходит наполнение множества этим значением (из списка допустимых значений). Такую операцию выполняет среда программирования. Однако включение во множество допустимой константы можно произвести и программно, например, после наступления какого-то события.

Дальнейшие пояснения проиллюстрируем на примере функции *POpenClick*. Если пользователь программы выбрал файл, расширение которого не совпадает с расширением, установленным (по умолчанию) в свойстве *DefaultExt*, то множественная константа этого свойства *ofExtensionDifferent* включается во множество *Options*, которое принадлежит объекту *OpenDialog1*.

Для проверки наличия во множестве конкретной множественной константы применяется метод *Contains* (подробнее о множествах см. раздел 13.2). Если попытаться открыть файл не с расширением *ole*, то метод *Contains* обнаружит в свойстве *Options* константу *ofExtensionDifferent*, после чего всплывёт окно с предупреждением *Invalid stream format* (недопустимый формат файла) и у пользователя появится возможность повторить операцию по выбору файла. В случае запуска файла с расширением *ole* выполняются те же операции, что и в предшествующем варианте функции *POpenClick*.

В функциях *PNewClick* и *POpenClick* можно опустить строку:

```
OleContainer1->DoVerb(ovShow);
```

В этом случае содержимое вставленного документа вы увидите на интерфейсе учебного приложения (или только часть документа, если его габариты превышают габариты интерфейса), однако программа, при помощи которой разработан документ, *автоматически* вызываться *не будет*. Для её запуска (если документ требуется отредактировать) следует сделать стандартный двойной щелчок по форме (фактически по контейнеру, на канве которого развёрнут выбранный вами документ). Иногда такой вариант бывает востребован. В исходном варианте со-

держимое вставленного документа также появляется на интерфейсе (канве контейнера), однако оно затем замещается окном программы (с этим же документом), автоматически вызываемой для обработки файлов соответствующего типа. Поэтому результат первого оператора программы трудно заметить.

Щелчок по подпункту *Заккрыть* обрабатывает функция *PCloseClick*:

```
void __fastcall TForm1::PCloseClick(TObject *Sender)
{
    OleContainer1->DestroyObject();
} //PCloseClick
```

Метод *DestroyObject()* разрушает объект, расположенный в контейнере *OleContainer1* (освобождает память, занятую объектом, а его указатель устанавливает в *NULL*). Здесь ещё раз следует напомнить о терминологии, которая в настоящее время утвердилась в языках визуального программирования. Визуальные и не визуальные объекты формы и *OLE*-объекты обсуждаемого выше контейнера на самом деле являются не объектами, а указателями на объекты, расположенные в динамической памяти (куче). Поэтому, когда утверждается: «метод *DestroyObject()* разрушает объект, расположенный в контейнере *OleContainer1*», на самом деле в контейнере обнуляется (устанавливается в *NULL*) указатель, направленный на объект динамической переменной (который уничтожается операцией *delete*).

Подраздел меню *Сохранить* реализует функция *PSaveClick*:

```
void __fastcall TForm1::PSaveClick(TObject *Sender)
{
    if (FName == "") //Если объект ранее не сохранялся
        if (SaveDialog1->Execute())
            FName= SaveDialog1->FileName;
    else
        return;
    OleContainer1->SaveToFile(ChangeFileExt(FName, ".ole"));
} //PSaveClick
```

Сохранение файла с внедрённым объектом *OLE* (в нужном вам каталоге) выполняется при помощи обычного диалога сохранения. В дальнейшем упомянутый файл можно открывать командой *Открыть* учебного приложения. Это полностью автономный объект, он не связан с исходным файлом, на основе которого был создан. Как отмечалось выше, открывать его можно только как объект *OLE*. Если имя файла ещё не задано (пустая строка), то вызывается диалог *SaveDialog1*, предназначенный для ввода имени (*Сохранить как*). После ввода имени объекта методом *SaveToFile(ChangeFileExt(FName, ".ole"))*, он сохраняется в файле с заданным именем и расширением *ole*. При повторных операциях сохранения, когда имя файла глобальной переменной *FName* уже присвоено (не пустая строка), диалог *SaveDialog1* не вызывается, а сразу же осуществляется обновление информации файла указанным выше методом.

В методе *SaveToFile* в качестве единственного аргумента вызывается функция *ChangeFileExt*: в данном конкретном вызове она *принудительно* устанавливает расширение *ole*. Это сделано во избежание ошибок программиста, для повыше-

ния надёжности работы программы: если программист в *Инспекторе Объектов* либо программно для упомянутых диалогов в свойстве *DefaultExt* не установил расширение «.ole». Функция *ChangeFileExt* возвращает имя файла *FName* (первый параметр функции) с изменённым расширением (второй параметр функции). В исходном файле (он был использован для внедрения его копии) расширение, конечно, не изменяется.

Теперь давайте посмотрим, как работает приложение в текущем варианте. Запустите его на выполнение, затем активизируйте подпункт *Новый*. В результате появится окно *Вставка объекта*, показанное на рис. 16.2.

Радиокнопка *Создать новый* позволяет (после нажатия кнопки *ОК*) создать новый *OLE*-документ при помощи выбранного приложения в окне *Тип объекта*. Если же активизировать радиокнопку *Создать из файла*, то увидите несколько другой вид окна с прежним заголовком, оно показано на рис. 16.3.

После нажатия командной кнопки с заголовком *Обзор* у вас появится возможность (с использованием стандартного окна) выбрать желаемый для вставки

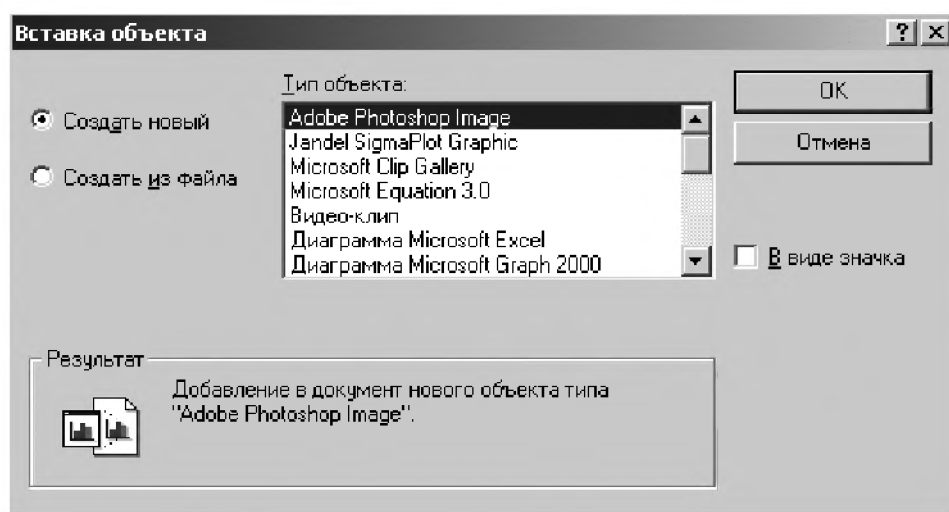


Рис. 16.2. Окно выбора приложения для вставляемого документа

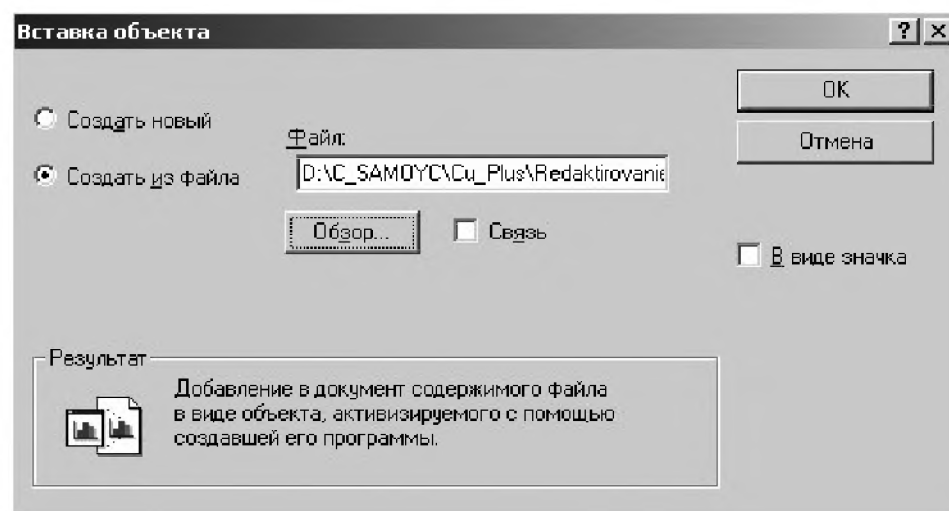


Рис. 16.3. Вид окна *Вставка объекта* при выборе радиокнопки *Создать из файла*

документ. Приложение, в котором он был разработан, в нашем варианте учебной задачи запустится автоматически (согласно расширению файла с документом). Если поставить галочку в индикаторе *В виде значка*, то вставленный документ будет заменён пиктограммой. Точно такой же индикатор с этими же функциями имеется и в окне, представленном на рис. 16.2.

Индикатор *Связь* позволяет осуществить связь с файлом. В этом случае в *OLE*-документ вставляется *только* адрес файла с документом (указатель). Поэтому все изменения документа-источника отображаются и в *OLE*-документе. Возможности по внедрению и связыванию документов точно такие же, как и в офисных приложениях *Windows*.

По умолчанию активизируется верхняя радиокнопка-переключатель *Создать новый*. Исследуем возможности этого режима. В окне *Тип объекта* выберите тип вставляемого в контейнер объекта. Тип объекта определяется приложением. Любое приложение, установленное на вашем компьютере, будет доступно в этом окне для выбора. Это может быть, например, *Word*, *Excel*, *Equation*, *Paint*, *SigmaPlot* и др. После выбора типа объекта щёлкните по кнопке *ОК*. В большинстве случаев в ваше приложение внедрится соответствующая программа вместе со своими инструментальными панелями и меню. На рис. 16.4 показана встроенная программа *Word*.

Обратите внимание, на интерфейсе вашего приложения *Пример OLE* появились почти все пиктограммы программы *Word*, а к пункту горизонтального меню с заголовком *Объект* добавились пункты горизонтального меню упомянутой программы. Вот только пункта *Файл* программы *Word* там нет. Как известно, в этом пункте находятся подпункты создания, открытия и сохранения файлов. Однако создание, открытие и сохранение *OLE*-файлов осуществляется разделами

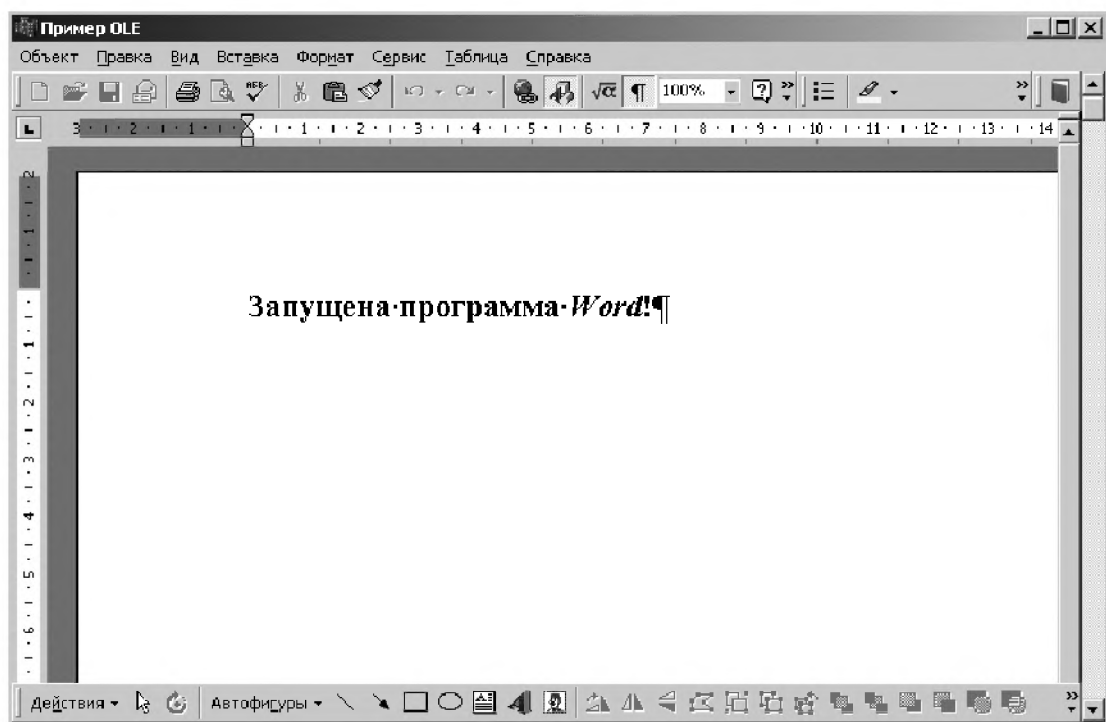


Рис. 16.4. Вид интерфейса приложения *Пример OLE* со встроенной программой *Word*

Новый, Открыть, Сохранить пункта *Объект* горизонтального меню *вашей* программы. Для указанных действий применяются методы, отличные от тех, которые используются в случае обработки файлов с другими расширениями. Ради иллюстрации такой чудесной трансформации в нашем учебном приложении и было сделано меню. По указанной выше причине три первые пиктограммы панели инструментов *Word* являются недоступными (*Новый, Открыть, Сохранить*). Следует обратить внимание на то, что, к сожалению, документы некоторых программ внедряются без своих инструментов. Например, окна документов *PowerPoint* внедряются без меню и пиктограмм. Причина этого будет рассмотрена несколько позже.

Теперь закройте документ *Word* (и его программу) при помощи подпункта *Заккрыть*. Далее вновь нажмите подпункт *Новый*, после чего в окне (см. рис. 16.2) включите радиокнопку *Создать из файла*, в появившемся новом окне (см. рис. 16.3) найдите и откройте, например, документ *Word, Excel* или *Paint*. Можно запустить даже файлы приложений операционной системы *DOS*, исполняемые файлы, например, с расширением *exe*. Документы упомянутых и многих других программ откроются вместе с соответствующими оболочками. Закрываются открытые документы при помощи пункта *Заккрыть*. Однако, если попытаться открыть, например, документ *PowerPoint*, то приложение зациклится, причину зависания компьютера обсудим позже.

После очередной активизации пункта *Новый* в появившемся окне включите индикатор *Связь*, при помощи кнопки *Обзор* найдите и откройте файл какого-либо доступного приложения, например, *Excel*. Теперь *вместо* учебного приложения на экране появится *только* интерфейс программы *Excel* с выбранным вами документом. После его редактирования и сохранения *все* внесенные изменения будут наблюдаться и в *OLE*-объекте, который связан с упомянутым файлом, и наоборот – все изменения в *OLE*-объекте отразятся в исходном файле.

В окне, возникающим при нажатии пункта *Новый*, при активизации любой из радиокнопок *Создать новый* или *Создать из файла* возможно включить переключатель *В виде значка*. В этом случае на форме появится значок вставленного документа только при отсутствии в функциях *PNewClick, POpenClick* строки операторов с методом *DoVerb*. Для вызова соответствующей программы следует выполнить двойной щелчок по контейнеру (по интерфейсу приложения). Если же метод *DoVerb* будет использоваться в указанных функциях, то переключатель *В виде значка* не будет влиять на работу учебного приложения.

16.1.2. Программное внедрение OLE-объектов

Выше была реализована возможность внедрения объекта *OLE* в результате диалога вставки объекта. Этот диалог позволяет выбрать и внедрить (или связать) любой доступный тип документа. Однако могут быть задачи, в которых тип внедряемого объекта *заранее* известен. В этом случае нет необходимости привлекать диалог вставки, разумнее использовать программное внедрение нужного объекта.

Для иллюстрации этой возможности в учебном приложении после пункта *Объект* добавьте пункты *MS-Word* и *MS-Excel*. Их вызов осуществляет запуск одноимённого приложения при помощи соответственно функций *MSWordClick* и *MSExeclClick*:

```
void __fastcall TForm1::MSWordClick(TObject *Sender)
{
    OleContainer1->CreateObject("Word.Document", false);
    OleContainer1->DoVerb(ovShow);
} //MSWord1Click

void __fastcall TForm1::ExcelClick(TObject *Sender)
{
    OleContainer1->CreateObject("Excel.sheet", false);
    OleContainer1->DoVerb(ovShow);
} //ExcelClick
```

В случае создания объектов иных типов второй оператор этих функций остаётся неизменным, а в вызове метода *CreateObject* меняется только первый параметр – класс создаваемого объекта *OLE*. Имена этих классов можно выяснить в справочниках либо в справочной системе *Builder*. Например, для запуска программы *PowerPoint* служит класс «*PowerPoint.showwt*». Однако запуск этого приложения у вас может не получиться, как не удалось это и нам. Дело в том, что *не любой* тип объекта и *не в любой* версии *Windows* может быть вставлен. Поэтому с переносимостью пользовательских приложений могут иногда возникать проблемы. Если вставка не может быть осуществлена, то, как правило, всплывёт окно с соответствующим сообщением. Однако компьютер иногда зависает без каких-либо предупреждений. Упомянутые причины объясняют указанные выше сбои в работе учебного приложения при попытках работы с документами и программой *PowerPoint*.

16.1.3. Быстрый способ внедрения и связывания объектов на основе существующих документов

Когда вы *заранее* знаете, что *OLE*-объект требуется внедрить (или связать (*Link*)) на основе уже *существующего* документа (заданного типа), то можно не вызывать окна, показанные на рис. 16.2 и 16.3. Более быстрый способ внедрения и связывания в учебном приложении реализуем при помощи введения дополнительных пунктов горизонтального меню с заголовками *Внедрить файл*, *Связать файл* и соответствующими именами *Insert_File*, *Link_File* (их следует добавить на интерфейсе приложения). В контейнере *OleContainer1* разместите ещё один диалог (*OpenDialog2*), его фильтр настройте на показ всех файлов либо в *Инспекторе Объектов*, либо программно при помощи кода:

```
OpenDialog2->Filter="Все файлы| *.*";
```

Эту строку кода можно, например, записать в конструкторе приложения *TForm1*, где уже имеются операторы по настройке фильтров диалогов, включённых в контейнер ранее. Функции, обрабатывающие нажатие вновь введенных

пунктов приводятся ниже.

```
void __fastcall TForm1::Insert_FileClick(TObject *Sender)
{
    if (OpenDialog2->Execute())
    {
        OleContainer1->CreateObjectFromFile(OpenDialog2->FileName, false);
        FName= OpenDialog2->FileName;
        OleContainer1->Repaint();
    } //if
} //Insert_FileClick

void __fastcall TForm1::Link_FileClick(TObject *Sender)
{
    if (OpenDialog2->Execute())
    {
        OleContainer1->CreateLinkToFile(OpenDialog2->FileName, false);
        FName= OpenDialog2->FileName;
        OleContainer1->Repaint();
    } //if
} //Link_FileClick
```

Внедрение и связывание в этих функциях выполняются при помощи методов контейнера:

```
void CreateObjectFromFile(String FileName, bool Iconic);
void CreateLinkToFile(String FileName, bool Iconic);
```

Аргумент *FileName* определяет имя файла, выбранного для внедрения или связывания. Если для второго аргумента установить значение *false*, то объект отображается в исходном виде (так, как он выглядит в файле), при значении *true* будет видна только пиктограмма объекта.

16.1.4. Сохранение документов в исходных форматах

Команда горизонтального меню *Связать файл* позволяет выполнять редактирование исходного файла в полноценной среде того приложения, в которой он был изначально разработан. В таких средах, безусловно, имеются и команды по сохранению документа. Поэтому в нашем приложении команда по сохранению связанного файла в его исходном формате (с первоначальным расширением) является излишней.

Вместе с тем, если открыть файл при помощи нового пункта *Внедрить файл* или с использованием команд *Объект/Новый*, *Объект/Открыть*, то вы увидите, что команды сохранения их родительских приложений заблокированы, неактивны. Все внесенные изменения можно сохранить в файле только с расширением *ole* при помощи команды *Объект/Сохранить*. Для сохранения изменений документа в файле с прежними расширениями (не с расширением *ole*) служит метод *SaveAsDocument*. Его использование иллюстрируется в функции *Save_FileClick*, предназначенной для обработки события, связанного с нажатием пункта с заголовком *Сохранить файл* (его имя *Save_File*), который также следует ввести в горизонтальное меню интерфейса приложения *Пример OLE*. В контейнере *OleContainer1* разместите ещё один диалог сохранения *SaveDialog2*.

```

void __fastcall TForm1::Save_FileClick(TObject *Sender)
{
    if (SaveDialog2->Execute())
    {
        OleContainer1->SaveAsDocument(SaveDialog2->FileName);
        FName= SaveDialog2->FileName;
    } //if
} //Save_FileClick

```

Интерфейс приложения *Пример OLE* в окончательном виде показан на рис. 16.5.

16.1.5. Развитие технологии OLE

Технология *OLE* появилась как *OLE 1.0* в операционной системе *Windows 3.1*. Дальнейшее развитие возможностей по использованию одних программ другими наблюдается в *OLE 2.0*. Основой этого усовершенствованного подхода явилась *компонентная модель объекта (COM, Component Object Model)*. Эта модель обеспечивает *полную* совместимость между компонентами (продуктами, приложениями, программами), написанными разными компаниями и на разных языках программирования, она предоставляет возможность одной программе (клиенту) работать с объектами другой программы (сервером).

Сервером может быть исполняемый файл или модуль с функциями, например библиотека *DLL* (динамически присоединяемая библиотека). Внешние приложения, обращающиеся к объекту *COM*, являются *клиентами COM*. Клиент получает указатель на интересующий его объект сервера и через этот указатель имеет возможность вызывать методы объекта. Разработчик клиентского приложения может получить информацию об объектах *COM* (свойствах, методах и параметрах методов) из библиотеки типов, которая создаётся разработчиком объекта *COM* и распространяется *вместе с объектом*.

Главное отличие *OLE 2.0* от *OLE 1.0* состоит в том, что компоненты (модули) доступны *только* программно, они создаются и используются при помощи *программного кода* и, следовательно, являются всегда *временными*. Они не могут

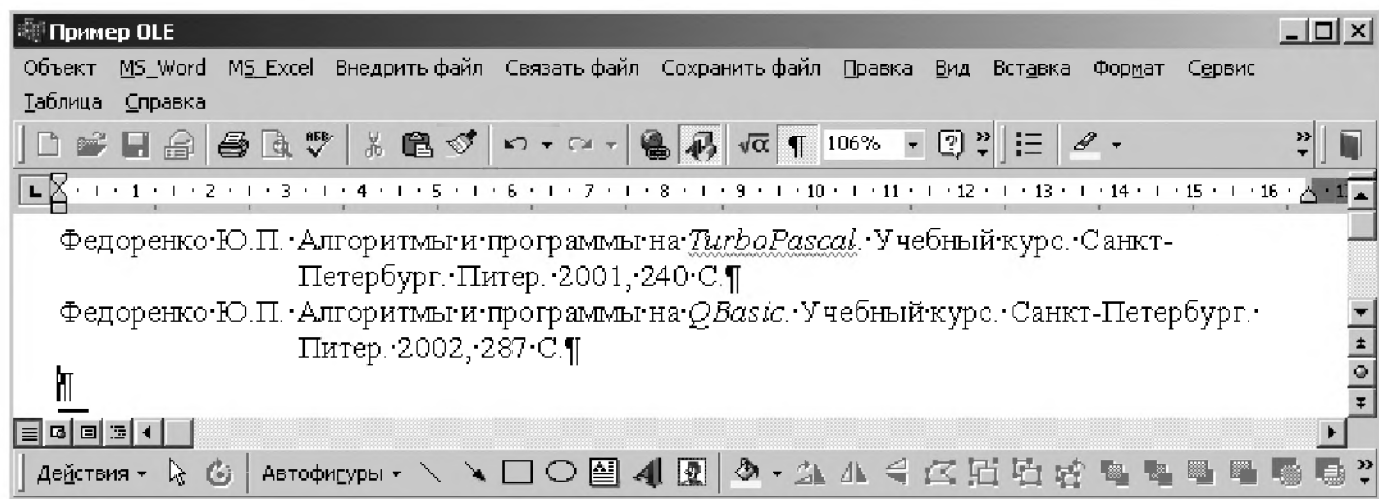


Рис. 16.5. Окончательный вид интерфейса приложения *Пример OLE* с внедрённым документом *Word*

быть внедрены или связаны, поскольку существуют только в течение времени выполнения поглотившей их программы, не видны непосредственно пользователю приложения.

Технология *COM* реализуется при помощи использования специальных библиотек, в частности, *OLE32.dll* и *OLEAut32.dll*. Они содержат стандартные интерфейсы и *API*-функции, обеспечивающие создание объектов *COM* и управление ими.

Следующим шагом развития взаимодействия клиентских и серверных программ является технология *CORBA* (*Common Object Request Broker Architecture*) – это стандарт построения приложений с *распределёнными объектами*. Программная реализация *CORBA* разработана для большинства платформ и с этой точки зрения универсальна. Она позволяет общаться друг с другом приложениям, созданным не только на разных языках программирования, но и работающим в разных операционных средах и на разных типах компьютеров. Взаимодействие клиентских и серверных приложений осуществляется с помощью специальных программ – объектных брокеров запросов *ORB* (*Object Request Broker*).

Сетевой агент *Smart Agent* является распределённой службой, хранящей информацию о запущенных в среде серверах. Эта служба запускается на одном из компьютеров сети и обеспечивает взаимодействие брокеров. Служба *Implementation Repository* хранит сведения обо всех зарегистрированных объектах *COM* и о местоположении ресурсов. Служба *Basic Object Adapter* обеспечивает регистрацию в среде серверов *CORBA* и их объектов.

Приведенный очень краткий перечень служб, особенностей технологий *COM* и *CORBA* позволяет получить о них лишь самые *общие* представления. Безусловно, в этих технологиях много общего в смысле конечных целей решаемых задач, однако пути решения этих задач *существенно* отличаются по сложности, которая, видимо, пропорциональна надёжности достигаемых решений. Подробное рассмотрение технологий *COM* и *CORBA* выходит за рамки настоящего пособия, является предметом отдельной книги.

Вопросы для самоконтроля

- ☐ Почему документ со связанным объектом необходимо передавать потребителю вместе с файлом, содержащим этот объект?
- ☐ В каких случаях используется метод *DoVerb* при вставке документов других программ?
- ☐ В приложении *Пример OLE* команда *Объект/Заккрыть* закрывает файлы с произвольным расширением, а не только с расширением *ole*. Как организовать обработку *дополнительного* пункта горизонтального меню с заголовком *Выход* уже существующей функцией *PCloseClick*?
- ☐ Будет ли работа приложения *Пример OLE* более эффективной, если программную настройку свойств диалогов осуществлять не в конструкторе, а в функциях, в которых они используются?

16.2. Задача для программирования

В приложение *Пример OLE* добавьте пункт горизонтального меню *exe-файлы*, вызов которого позволит находить и запускать исполняемые файлы с расширением *exe*.

16.3. Вариант решения задачи

Код функции по обработке вызова пункта горизонтального меню с заголовком *exe-файлы* и именем *File_exe*:

```
void __fastcall TForm1::File_exeClick(TObject *Sender)
{
    //Настройка свойств диалога OpenFileDialog
    OpenFileDialog->DefaultExt= "exe";
    OpenFileDialog->Filter="exe-файлы|*.exe|Все файлы| *.*";
    if (OpenDialog3->Execute())
    {
        OleContainer1->CreateObjectFromFile(OpenDialog3->FileName, false);
        FName= OpenFileDialog->FileName;
        OleContainer1->Repaint();
    }
}
//File_exeClick
```




Список литературы

1. *Архангельский А.Я.* Программирование в C++ Builder 6. – М.: БИНОМ, 2002. – 1151 с., ил.
2. *Архангельский А.Я.* Справочное пособие. Книга 1. Язык C++. – М.: БИНОМ, 2002. – 543 с., ил.
3. *Архангельский А.Я.* Справочное пособие. Книга 2. Классы и компоненты. – М.: БИНОМ, 2002. – 526 с., ил.
4. *Бобровский С.* Самоучитель программирования на языке C++ в системе Borland C++ Builder 5. – М.: ДЕСС, I-PRESS, 2001. – 272 с., ил.
5. *Подбельский В.В.* Язык СИ. Учебное пособие. – М.: Финансы и статистика, 2001. – 559 с., ил.
6. *Павловская Т.А.* C/C++. Программирование на языке высокого уровня. Учебник. – СПб.: Питер, 2001. – 460 с., ил.
7. *Франка П.* C++. Учебный курс. – СПб.: Питер, 2001. – 521 с., ил.
8. *Страуструп Б.* Язык программирования C++. – СПб.: Невский Диалект – М.: БИНОМ, 1999. – 991 с., ил.
9. *Болгарский Б.В.* Очерки по истории математики. – Минск: Высшая школа, 1974. – 287 с., ил.
10. *Мюрей У., Папас К.* Visual C++. Руководство для пользователей. – СПб.: BHV, 1996. – 912 с., ил.
11. *Галявов И.Р.* Borland C++5 для себя. Самоучитель. – М.: ДМК Пресс, 2001. – 427 с., ил.
12. *Холлинг Д., Баттерфилд Д., Сворт Б. и др.* Руководство разработчика. Т. 1, Основы. – Москва-Петербург-Киев: Издательский дом «Вильямс», 2001. – 872 с., ил.
13. *Шамис В.А.* Borland C++ Builder 6 для профессионалов. – СПб.: Питер, 2003. – 797 с., ил.
14. *Культин Н.* Самоучитель C++ Builder. – СПб.: БХВ-Петербург, 2004. – 320 с., ил.
15. *Боровский А.* Программирование в Delphi. – СПб.: БХВ-Петербург, 2005. – 448 с., ил.
16. *Секунов Н.* Самоучитель Visual C++.NET. – СПб.: БХВ-Петербург, 2002. – 736 с., ил.
17. *Лабор В.В.* Си Шарп. Создание приложений для Windows. – Минск: Харвест, 2003 г. – 384 с., ил.
18. *Черносвитов А.* Visual C++7. Учебный курс. – СПб.: Питер, 2002. – 528 с., ил.
19. *Якушев Д.* Философия программирования на языке C++. – М.: Бук-пресс, 2006. – 320 с. ил.
20. *Архангельский А.Я., Тагин М.А.* Программирование в C++ Builder 6 и 2006. – М.: Бином-Пресс, 2007. – 1181 с., ил.
21. *Архангельский А.Я.* Приёмы программирование в C++ Builder 6 и 2006. – М.: Бином-Пресс, 2006. – 992 с., ил.



Алфавитный указатель компонентов библиотеки **VCL**

A

Animate 132

B

Button 28

C

CGauge 311

Chart 117

CheckBox 304

ColorDialog 319

E

Edit 26

FontDialog 319

G

GroupBox 303

I

Image Editor 254

L

Label 27

M

MainMenu 248

Memo 229

O

OpenPictureDialog 477

P

Picture Editor 252

ProgressBar 311

R

RichEdit 229

S

ShortCut 251

SpeedButton 252

StaticText 311

StringGrid 90

T

TBitBtn 88, 134

TBitmap 480

TButton 88

TIcon 480

TMetafile 480

ToolBar 252

TStringList 232

U

UpDown 94



Алфавитный указатель функций библиотек **C++Builder**, **API Windows**, директив компилятору, типов данных, операций и других ключевых слов

__published 414

A

abs 75

acos 74

AddXY 123

Arc 489

asin 74

atan 74

B

Beep() 131

Bounds 487

break 87

C

c_str() 341

catch 51

ceil 74

char 329

CharToOem 333

CharToOemBuff 333

Clear() 123

Close() 131

Contains 126

CopyRect 506

cos 74

CurrToStr 340, 341

CurrToStrF 340, 341

D

Delete 343

delete 197, 264, 281

do_while 105

double 73

DWORD 312

E

EConvertError 50

Ellipse 488

exit 132

ExitWindowsEx 311

F

fabs 75

FileOpen 242

FileRead 242

FileSeek 242

FileWrite 242

FillRect 487

float 32

FloatToStr 36

FloatToStrF() 48

FloatToStrF() 47

floor 74

fmod 75

for 96

G

GetTickCount() 312

H

heap 261

I

if 80

include 71

Insert 343

int 76

IsDelimiter 346

IsEmpty() 344

L

labs 75

LineTo 490

log 74

long double 74

long int 75

LowerCase 344

M

MoveTo 490

N

NAN 50

new 197, 263, 279

O

OemToChar 331

OemToCharBuff 331

P

Pie 489

Play 133

Polygon 492

Polyline 491

Pos 342

pow 74

private 393

protected 393

public 393

R

Rect 487

return 201

S

Set 365

SetLength 344

ShowMessage() 51

sin 74

sqrt 74

stdlib.h 75

strcat 336

strcpy 336

String 237, 339

StringOfChar 345

StringReplace 345

strlen 337

strstr 337

strtok 237, 338

struct 376

SubString 345

switch 84

T

tan 74

template 208

Terminate() 132

TextOutA 493

TextRect 487

this 457

ToDouble 340

ToInt 340

ToIntDef 340

Trim() 344

TrimLeft 344

TrimRight 344

try 51

U

UndoZoom() 126

union 374

unsigned char 330

UpperCase 344

W

Warning 43, 61

Warnings 60

while 103

WrapText 346